BinLPT: A Workload-Aware Parallel Loop Scheduler for Large-Scale Multicore Platforms

Pedro Henrique Penna

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Pedro H. Penna

BINLPT: A WORKLOAD-AWARE PARALLEL LOOP SCHEDULER FOR LARGE-SCALE MULTICORE PLATFORMS

Florianópolis

2017

Pedro H. Penna

BINLPT:
A WORKLOAD-AWARE PARALLEL LOOP SCHEDULER FOR LARGE-SCALE MULTICORE PLATFORMS

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Ciência da Computação para obtenção do Grau de Mestre em Ciência da Computação.

Orientador: Prof. Márcio Castro
Universidade Federal de Santa Catarina

Coorientadora: Prof^{a}. Patricia Plentz
Universidade Federal de Santa Catarina

Florianópolis

2017

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Penna, Pedro Henrique
BinLPT : A Workload-Aware Parallel Loop Scheduler for Large-Scale Multicore Platforms / Pedro Henrique Penna ; orientador, Márcio Castro, coorientadora, Patricia Plentz, 2017.
107 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós Graduação em Ciência da Computação, Florianópolis, 2017.

Inclui referências.

1. Ciência da Computação. 2. Computação de Alto Desempenho. 3. Aplicações Irregulares. 4. Escalonamento de Laços. I. Castro, Márcio. II. Plentz, Patricia. III. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. IV. Título.

Pedro Henrique de Mello Morado Penna

BinLPT: A Workload-Aware Parallel Loop Scheduler for Large-Scale Multicore Platforms

Esta dissertação foi julgada adequada para obtenção do título de mestre e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Florianópolis, 1º de agosto de 2017.

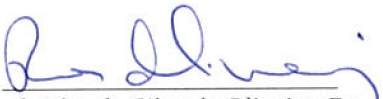
Prof. José Luís Almada Güntzel, Dr.
Coordenador do Programa

Profª. Patricia Della Méa Plentz, Drª.
Universidade Federal de Santa Catarina
Coorientadora

Banca Examinadora:

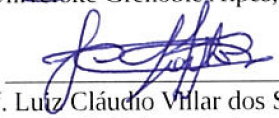
Prof. Márcio Bastos Castro, Dr.
Universidade Federal de Santa Catarina
Orientador

p/ Prof. Henrique Cota de Freitas, Dr.
Pontifícia Universidade Católica de Minas Gerais
(Vídeoconfeferência)

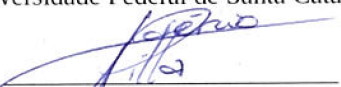


Prof. Rômulo Silva de Oliveira, Dr.
Universidade Federal de Santa Catarina

Prof. Jean-François Méhaut, Dr.
Université Grenoble Alpes, França



Prof. Luiz Cláudio Villar dos Santos, Dr.
Universidade Federal de Santa Catarina



Prof. Laércio Lima Pilla, Dr.
Universidade Federal de Santa Catarina

RESUMO

A comunidade de Computação de Alto Desempenho investiga soluções eficientes e escaláveis, visando suprir as demandas computacionais de aplicações larga-escala. Para tanto, particularidades da aplicação e plataforma alvos são exploradas, de forma que técnicas específicas possam ser aplicadas. Nesse contexto, a irregularidade da aplicação é uma importante característica que deve ser considerada, por exemplo, no escalonamento de iterações de laços. Nesse cenário, estratégias cientes da carga de trabalho destacam-se como a abordagem mais promissora, no entanto elas apresentam algumas fraquezas que devem ser superadas. Primeiro, elas baseiam-se em técnicas de *profiling* e regressão estatística, sendo portanto inerentemente projetadas para cargas de trabalho com padrão bem definido. Segundo, essas estratégias falham em aplicar seu conhecimento para escalonar os *chunks* de iterações do laço paralelo alvo. Terceiro, estratégias existentes não foram avaliadas de maneira compreensiva quanto a variações na carga de trabalho. Finalmente, apesar de existirem diversas estratégias cientes da carga de trabalho, nenhuma delas está integrada a uma biblioteca de programação paralela, tornando assim ainda mais desafiador que aplicações beneficiem-se das mesmas. Com o objetivo de endereçar esses problemas, nesse trabalho propõe-se uma nova estratégia de escalonamento de laços ciente da carga de trabalho batizada de BinLPT. Para possibilitar desempenho e flexibilidade superiores, essa estratégia é baseada em algumas características como estimativas da carga de trabalho fornecidas pelo usuário e o uso de uma heurística de escalonamento adaptativa baseada na regra LPT. Além disso, o BinLPT foi integrado no OpenMP e disponibilizado publicamente para *download*. Essa nova estratégia foi concebida com base em dois pontos, ambos desenvolvidos durante a elaboração dessa dissertação: uma metodologia de projeto para novas estratégias de escalonamento de laços baseado em simulação; e uma estratégia prova de conceito, nomeada SRR. Uma avaliação compreensiva do BinLPT foi efetuada, utilizando simulações, *benchmarks* sintéticos e *kernels* de aplicações, com diversas cargas de trabalho. Os experimentos foram executadas em uma plataforma NUMA e os resultados com os *kernels* de aplicações relevaram que BinLPT conduz a um desempenho de até 64.92% superior que estratégias do OpenMP consideradas.

Palavras-chave: Computação de Alto Desempenho. Aplicações Irregulares. Estratégias de Escalonamento de Laços.

ABSTRACT

The High Performance Computing community seeks for efficient and scalable solutions to meet the ever-increasing performance demands in large-scale applications. To achieve this goal, intricacies of the target application and platform are often exploited, so that specific techniques can be applied. In this context, the irregularity of the application is an important characteristic that should be considered. For instance, when scheduling loop iterations of a shared-memory-based applications, workload-aware scheduling strategies stands out as the most promising approach. Unfortunately, existing strategies that are based on this finding present several drawbacks that should be overcome. First, these strategies rely on profiling and statistical regression techniques, and thus are inherently designed to well-behaved workloads. Second, workload-aware strategies fail to apply their knowledge about the underlying workload of the target irregular loop when scheduling chunks of iterations. Third, existing strategies were not so far comprehensively evaluated in what concerns variations in the workload. Finally, despite the existence of several workload-aware strategies, none of them is integrated in a publicly available library for parallel programming, hence making it harder for applications to effectively get benefit from them. To address these challenges, in this work we propose a novel workload-aware loop scheduling strategy called BinLPT. To enable superior performance and flexibility, our strategy is based on some features as user-supplied estimation about the workload of the target irregular loop and the use of an adaptive scheduling heuristic based on the Longest Processing Time (LPT) rule. We integrated BinLPT into OpenMP, and we made our implementation publicly available. To conceive BinLPT, we relied on two cornerstones, both devised during the preparation of this master thesis: a simulation-guided design methodology and on a proof-of-concept workload-aware loop scheduler, named Smart Round-Robin (SRR). We carried out a throughout assessment of BinLPT using simulations, synthetic kernels and application kernels. We ran experiments on a large-scale NUMA machine and we studied the different workloads. In the application kernels, our experimental results uncovered up to 64.91% of performance improvement when using BinLPT, in contrast to OpenMP strategies.

Keywords: High Performance Computing. Irregular Applications. Loop Scheduling Strategies.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

LS	List Scheduling
LPT	Longest Processing Time
PSS	Pure Static Scheduling
PDS	Pure Dynamic Scheduling
CSS	Chunk Self-Schedling
GSS	Guided Self-Scheduling
FSS	Factoring Self-Scheduling
TSS	Trapezoid Self-Scheduling
AFS	Affinity Self-Scheduling
AWF	Adaptive Weighted Factoring
HSS	History-Aware Self-Scheduling
KASS	Knowledge-Based Adaptive Self-Scheduling
SRR	Smart Round-Robin
SPEC	Standard Performance Evaluation Corportation
NPB	NAS Parallel Benchmarks
CFD	Computational Fluid Dynamics
GCC	GNU Compiler Collection
API	Application Programming Interface
IID	Independent and Identically Distributed
CoV	Coefficient of Variance
PDF	Probability Density Function
HPC	High Performance Computing
SMT	Simultaneous Multithreading
NUMA	Non-Uniform Memory Access
SMP	Symmetric Multiprocessing
AMP	Assymmetric Multiprocessing
GA	Genetic Algorithm

CONTENTS

1 INTRODUCTION

The High Performance Computing (HPC) community constantly works on the design of efficient and scalable alternatives, ranging from hardware architecture (DINECHIN et al., 2013) to runtime support (BROQUEDIS et al., 2010), to meet the ever-increasing cutting-edge performance demands in large-scale engineering and scientific applications. Indeed, these solutions are often tailored to address the particular intricacies of an application and its underlying platform, so that specialized techniques can be employed (FRANCESQUINI et al., 2015).

In this context, the irregularity of an application is an important feature that affects its performance, and hence it is often used to classify parallel applications into two groups: *regular applications* and *irregular applications* (GAUTIER; ROCH; VILLARD, 1995). In the first group, the amount of computation that is required to solve a given problem depends only on the size of the input data. An implementation of the Naive Matrix Multiplication Algorithm for dense matrices is a typical example of a highly regular application. In this particular application, the number of floating point operations is constantly proportional to the matrix size, regardless the actual numbers involved in the computation. On the other hand, in irregular applications, the contents of the input data also impact the execution time significantly. For instance, in N-Body Simulations, the number of particle interactions to be computed depends on both the number of particles in the physical system and the spatial distribution of those particles.

Although subtle, the difference between these two groups directly impacts the design of efficient parallel solutions. Indeed, regular applications are especially preferred because their workload can be trivially broken up into homogeneous tasks, by simply dividing the total workload by the number of working threads of the application. With such strategy, each thread is assigned to an even amount of load, and thus optimal performance is achieved. Unfortunately, in irregular applications, this strategy may potentially lead to a set of tasks which are heterogeneous in terms of load, thus causing load imbalance among the threads. Consequently, the overall execution time is bounded by the performance of the most overloaded thread, which in turn creates scalability problems.

1.1 MOTIVATION

Evenly distributing the workload among the threads of an irregular application is an NP-Hard minimization problem known as the Multiprocessor Scheduling Problem (GRAHAM, 1969; GAREY; JOHNSON, 1979). This problem presents a significant challenge to both academic and industry communities, and it is a recurring subject of research in HPC. For instance, in shared-memory-based parallel applications, this problem emerges when scheduling iterations of irregular parallel loops with no dependency (FANG et al., 1990; POLYCHRONOPOULOS; KUCK, 1987; HUMMEL; SCHONBERG; FLYNN, 1992). In this scenario, the problem is referred as the Loop Scheduling Problem, and it can be reduced to the assignment of independent loop iterations such that (a) their load is evenly distributed, and thus execution time reduced; and (b) the scheduling overhead is minimized.

Several loop scheduling strategies have been proposed to address the previous problem (KRUSKAL; WEISS, 1985; FANG et al., 1990; POLYCHRONOPOULOS; KUCK, 1987; HUMMEL; SCHONBERG; FLYNN, 1992; TZEN; NI, 1993; MARKATOS; Le Blanc, 1994; HURSON et al., 1997), and they mainly rely on two techniques. In the first one, called on-demand scheduling, iterations are scheduled to threads on-the-fly at runtime, so that both load imbalance and runtime variations may be dynamically handled. In the second technique, called chunk-size tuning, iterations are scheduled in optimally sized batches (*i.e.* chunks) so that (i) scheduling overheads are mitigated, (ii) load imbalance is further amortized and (iii) iteration affinity is exploited. When coupled together, on-demand scheduling and chunk-size tuning may indeed deliver reasonable performance to a wide range of scenarios. Nevertheless, these techniques do not consider any knowledge about the underlying workload of the target parallel loop, and thus scheduling strategies built upon them naturally turn out to be suboptimal (BALASUBRAMANIAM et al., 2012; PENNA et al., 2016).

To address this limitation, workload-aware strategies were introduced (BANICESCU; Flynn Hummel, 1995; BULL, 1998; BANICESCU; VELUSAMY, 2001; KEJARIWAL; NICOLAU; POLYCHRONOPOULOS, 2006; WANG et al., 2012). These strategies rely on some workload knowledge to adaptively fine-tune chunk sizes, and thus further amortize load imbalance and deliver superior performance.

1.2 LIMITATIONS OF EXISTING STRATEGIES

Although workload-aware strategies present better performance gains than workload-unaware strategies (or blind strategies), they still face some drawbacks that should be tackled. First, these strategies rely on profiling and statistical regression techniques, and thus are inherently designed to well-behaved workloads. Therefore, to tackle irregular loops in which the workload varies drastically, some alternative for estimating the workload on-the-fly is necessary. Furthermore, it is worth noting that the loop scheduling strategy itself and the workload-estimation technique should be loosely coupled. This way, HPC engineers may plug into their solutions the workload-estimator that best fits their needs. Unfortunately, however, existing knowledge-based strategies do not provide this flexibility.

Second, workload-aware loop scheduling strategies fail to apply their knowledge about the underlying workload of the target irregular loop when scheduling chunks of iterations. Pragmatically, workload-aware strategies rely on the on-demand scheduling technique, and thus inherently achieve suboptimal performance when the workload estimation is accurate (GRAHAM, 1969). Furthermore, on-demand scheduling is known to lead to scalability problems (BULL, 1998), and thus should be wisely avoided when designing applications for exascale platforms.

Third, existing workload-aware loop scheduling strategies were not so far comprehensively evaluated in what concerns variations in the workload. This lack in performance analysis is due to the poor availability of an evaluation methodology and benchmark programs to target this particular scenario. Moreover, this limited assessment undertakes a throughout understanding of the lower and upper bound potentials of workload-aware scheduling strategies.

Finally, despite the existence of several workload-aware strategies, none of them is integrated in a publicly available Application Programming Interface (API), library or framework for parallel programming, such as OpenMP, TBB or Cilk. Indeed, the integration of these strategies with an irregular application is not trivial (BANICESCU; Flynn Hummel, 1995; BANICESCU, 2003), thereby restricting their use to an even smaller subset of irregular applications.

1.3 GOALS & CONTRIBUTIONS

Targeting the problems stated previously, the main goal of this work is to propose a novel workload-aware loop scheduling strategy for irregular parallel loops with no dependency. This new strategy overcomes the aforementioned weaknesses in workload prediction and chunk-scheduling that limit the performance of existing strategies. Moreover, we aim at delivering a publicly available implementation of the proposed loop scheduling strategy in a widely-used API for parallel programming. Therefore, this work delivers the following main contributions to the state-of-the-art:

- *A novel workload-aware loop scheduling strategy entitled BinLPT.* To enable superior performance and flexibility, our strategy is based on three features. First, it relies on some user-supplied estimation about the workload of the target irregular loop. Such estimation may be retrieved either from the problem structure or through online/offline profiling, thus enabling maximum flexibility. Second, BinLPT uses a greedy bin packing heuristic to adaptively partition the iteration space into several chunks. The maximum number of chunks that may be produced is a parameter of our strategy, and it may be fine-tuned to better meet the characteristics of the irregular parallel loop. Third, BinLPT schedules chunks to threads using a hybrid scheme based on the LPT rule and on-demand scheduling (GRAHAM, 1969), thereby ensuring that load imbalance and runtime variations are optimally handled. Existing strategies pragmatically rely only on the latter rule.

- *An integration of BinLPT into the OpenMP runtime system of GCC.* OpenMP is a parallel programming API for shared-memory architectures that is widely-used by the academic and industry communities in HPC (DAGUM; MENON, 1998). Our implementation is open-source and publicly available for download, thereby enabling any parallel application that relies on this programming abstraction to seamlessly use our strategy.

Furthermore, to deliver a comprehensive performance evaluation of BinLPT, we carried out a throughout assessment using simulations, synthetic kernels and application kernels. We ran experiments on a large-scale Non-Uniform Memory Access (NUMA) machine and we studied the performance of BinLPT under a variety of irregular workloads.

To conceive BinLPT, we relied on two research works, both devised during the preparation of this master thesis: a design methodology for workload-aware loop scheduling strategies based on simulation (PENNA et al., 2016); and (ii) a proof-of-concept draft workload-aware loop scheduler named SRR (PENNA et al., 2017).

1.4 WORK ORGANIZATION

The remainder of this work is organized as follows. In Chapter 2, we present and discuss the underlying concepts on which this work relies. In Chapter 3, we present the related work on loop scheduling strategies, highlighting those works on workload-aware approaches. In Chapter 4, we detail the workload-aware loop scheduling strategy proposed in this work. In Chapter 5, we present our evaluation methodology. In Chapter 6, we discuss experimental results. In Chapter 7, we draw the conclusions of our work, and in Chapter 8 we discuss some valuable future works derived from ours.

2 BACKGROUND

In this chapter we discuss the background on which this work relies. First, we present a formal definition of the Loop Scheduling Problem. Then, we discuss the known bounds on multiprocessor scheduling. Next, we introduce the existing classical loop scheduling strategies. Finally, we give a background on loop scheduling support in OpenMP.

2.1 THE LOOP SCHEDULING PROBLEM

The Loop Scheduling Problem is an instance of the NP-Hard minimization problem for multiprocessor scheduling (GRAHAM, 1969; GAREY; JOHNSON, 1979) and can be stated as follows.

Definition. Let $\widehat{x} = (i_1, i_2, \ldots, i_m)$ be a sequence of m independent loop iterations, and $w_k \in \mathbb{N}^+$ be the load of iteration i_k. If $\widehat{c_k}$ is an arbitrary subsequence in $\widehat{x}$ (*i.e.* chunk of iterations), its load can be expressed as :

$$\omega(\widehat{c_k}) = \sum_{i_j \in \widehat{c_k}} w_j$$

Thus, given two arbitrary and non-overlapping chunks in $\widehat{x}$, say $\widehat{c_a}$ and $\widehat{c_b}$, the load imbalance between them is:

$$\psi(\widehat{c_a}, \widehat{c_b}) = \mid \omega(\widehat{c_a}) - \omega(\widehat{c_b}) \mid$$

Moreover, given a set of non-overlapping chunks C_a in $\widehat{x}$, the load imbalance of C_a in respect to $\widehat{x}$ is (Equation 2.1):

$$\varphi(C_a, \widehat{x}) = \mid \sum_{\widehat{c_i} \in C_a} \omega(\widehat{c_i}) - \frac{\omega(\widehat{x})}{m} \mid \tag{2.1}$$

Therefore, the Loop Scheduling Problem comes down to partitioning $\widehat{x}$ into n non-overlapping chunks and assigning disjoint sets of these n chunks to the p threads; such that the following are minimized:

i. the number of n chunks that are used to partition the loop iteration space $\widehat{x}$;

ii. and the load imbalance of the most overloaded thread $\varphi(C_{\max}, \widehat{x})$ (Equation 2.1). ■

The previous formulation depicts the relation of the four core variables of the Loop Scheduling Problem: (i) the loop iteration space $\widehat{x}$; (ii) the load of iterations w_k; (iii) the number of chunks n in which $\widehat{x}$ will be partitioned; and (iv) the number of threads p that will process the n chunks in parallel. However, additional variables should be considered when the problem is analyzed in a real-world context. Therefore, in the paragraphs that follow, we discuss some of the most significant ones.

Scheduling Overhead This is an important concern for strategies that assign chunks of iterations to idle threads on-the-fly. If contention in synchronization structures is costly and the on-demand scheduling strategy is frequently invoked, the irregular parallel loop may face severe scalability issues when the number of chunks grows asymptotically (FANG et al., 1990).

Memory Affinity It is related to the temporal and spatial data localities that exist across the iteration space. When memory affinity [1] is exploited, the memory hierarchy is efficiently used, thereby reducing contention in buses and other interconnection structures and thus increasing performance. This variable greatly impacts the performance of memory-intensive irregular loops (MARKATOS; Le Blanc, 1994). Memory affinity is likely to be exploited when using large chunks.

Platform Heterogeneity It is related to the processing heterogeneity of the underlying platform. In Assymmetric Multiprocessing (AMP) machines, the processing capacity and features of each processing unit may greatly differ from one another, and thus impact the overall chunk scheduling performance (CHEN et al., 2012). Fortunately, however, such architectural characteristics can be queried at runtime and thus may be handled in a per-platform fashion.

Platform Availability It is related to the presence of external load running concurrently on the platform, such as jobs of other users. Unlike architecture heterogeneity, platform availability is unpredictable, which in turn makes it harder to deal with (BANICESCU, 2003).

[1] Data and temporal locality

2.2 KNOWN BOUNDS ON LOOP SCHEDULING

The Loop Scheduling Problem is an instance of the Multiprocessor Scheduling Problem, which in turn is a classical problem in Computer Science and has been extensively studied so far. Based on this observation, in this section, we carry out a discussion on the known bounds for the multiprocessor scheduling aiming at uncovering the bounds on loop scheduling as well. The following analysis considers the core variables that were introduced in the previous section, and it assumes that threads have equal processing capacities.

The lower bound solution for the Multiprocessor Scheduling Problem is to assign to each thread a workload that equals the overall workload divided by the number of threads. Let $X = \{u_1, u_2, \ldots, u_n\}$ be a set of n independent tasks, $w_k \in \mathbb{N}^+$ be the load of iteration u_k, and p the number of threads, the optimum amount of workload W^* to assign to each thread is (Equation 2.2):

$$W^* = \frac{\sum_{k=1}^{n} w_k}{p} \tag{2.2}$$

This solution may not be achievable in several instances of the Multiprocessor Scheduling Problem, because there may not even exist a task scheduling that would lead to such an optimum workload distribution. Nevertheless, it can be used as a baseline for evaluating scheduling strategies. In the Loop Scheduling Problem, for instance, an analogous optimum strategy would be able to evenly distribute the underlying workload of a target parallel loop.

On the other hand, an upper bound solution for the Multiprocess Scheduling Problem can be obtained by the following greedy algorithm (GRAHAM, 1969): consider tasks in arbitrary order, and assign them on-demand to idle threads. This strategy is known as the List Scheduling (LS) strategy and leads to a 2-approximation solution.

Proof. Let W^* be the optimum workload, let W_i be the overall workload assigned to the most overloaded thread i, and let j be the last iteration assigned to this thread. Note that, before j was assigned to i, thread i had the smallest load. Therefore, we have:

$$W_i = \underbrace{(W_i - w_j)}_{\leq W^*} + \underbrace{w_j}_{\leq W^*} \leq 2W^*$$

■

Indeed, a tight bound solution for the Multiprocessing Scheduling Problem was introduced by Graham (1969) and it is known as the LPT strategy. This strategy leads to a 4/3-approximation of the optimum solution and works similarly to LS, but it considers tasks in decreasing order of load instead of in arbitrary order. We refer the reader to Graham (1969) for the proof of this notable outcome.

Reasoning on these two latter bounds, the following conclusion may be derived for the Loop Scheduling Problem. Strategies that perform fine-grain on-demand scheduling are bounded to a 2-approximation solution. However, if the underlying workload of the target parallel loop is sorted in a quasi-decreasing fashion, then a 4/3-approximation solution will follow.

The previously conclusion holds when the scheduling overhead incurred by fine-grain scheduling is negligible. Unfortunately, this is not true in practice, and thus fine-grain on-demand loop schedulers may present worse performance that theoretically they would output. Nevertheless, Kruskal and Weiss (1985) and Hummel, Schonberg and Flynn (1992) showed that if the number of iterations in the parallel loop is asymptotically large, medium-grain on-demand scheduling yields 2-approximation solutions. In their original works, they consider a *batch of loop iterations* to be a set of iterations that are assigned in a single round. If for each batch that is scheduled there is more than a half of iterations left to be scheduled in subsequent batches, there exists a high probability that all threads will end up receiving an even portion of the overall workload, regardless the Probability Density Function (PDF) associated to the load of iterations. Moreover, this probability increases as the ratio between number of loop iteration and the number of threads increases. Figure 1 illustrates this outcome, which we refer

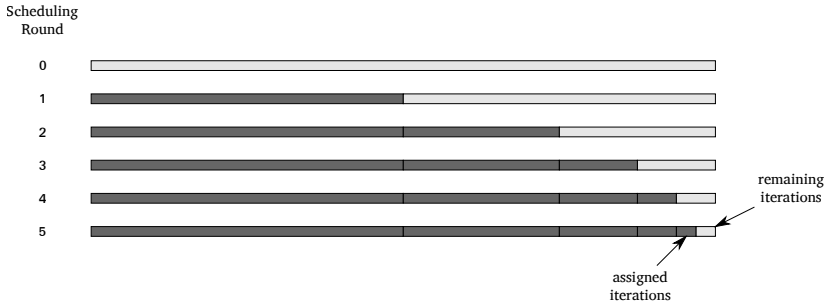


Figure 1: Load imbalance amortization principle.

in this work as the "*Load Imbalance Amortization Principle*". In this example, half of the remaining iterations are assigned to threads at each scheduling round.

2.3 CLASSICAL LOOP SCHEDULING STRATEGIES

Loop scheduling strategies boil down to one of the following two approaches: *static*, in which loop iterations are assigned to the threads of the parallel application at compile-time; and *runtime*, in which scheduling decisions are made at runtime (KEJARIWAL; NICOLAU; POLYCHRONOPOULOS, 2006). Static scheduling strategies introduce no runtime overhead, but (i) they are only possible on parallel loops which can have their bounds somehow determined at compile time, and (ii) they are suitable only for parallel loops which feature a compile-time predictable workload. In contrast, runtime strategies are employed to address parallel loops that either do not meet the aforementioned compile-time requirement, or perform computation on a workload that is known only at runtime. In this work, we address the problem of scheduling irregular parallel loops in which the workload is known only at runtime. In the following paragraphs we discuss the most important runtime scheduling strategies that have been widely-used so far, due to their applicability and performance on a great number of applications. Figure 2 illustrates the scheduling of each of these strategies when two threads are used, named A and B. In this figure, each cell represents an iteration of the parallel loop. Grey cells were scheduled to thread A and white cells to thread B

Pure Static Scheduling (PSS) It statically partitions a parallel loop into even-sized chunks of iterations, and then it assigns these chunks to threads, one at a time, statically and in a round-robin-fashion (Figure 2a). This strategy leads to minimum runtime overhead and thereby enables optimal performance for regular loops. However, in irregular loops, it may lead to poor load balancing and performance.

Pure Dynamic Scheduling (PDS) It assigns iterations to threads in unit-sized chunks and on-demand. Whenever a thread becomes idle, an iteration is assigned to it (Figure 2b). This strategy relies on the LS Rule (recall Section 2.2), and thus achieves good load balancing but at the price of a possibly overwhelming runtime overhead.

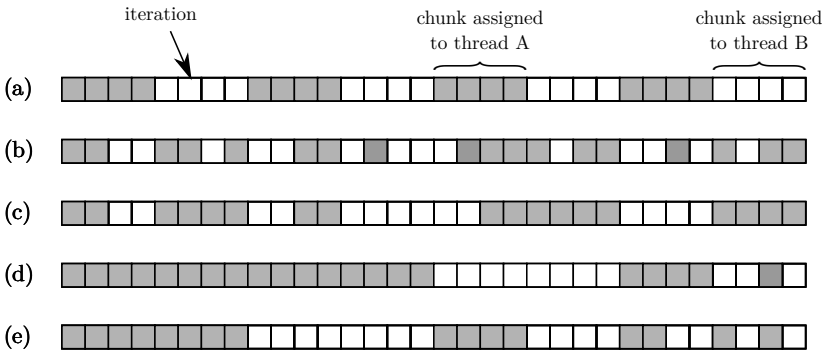


Figure 2: Classical loop schedulers in action: (a) Pure Static Scheduling (PSS) with chunk size 4; (b) Pure Dynamic Scheduling (PDS); (c) Chunk Self-Scheduling (CSS) with chunk size 2; (d) Guided Self-Scheduling (GSS); and (e) Factoring Self-Scheduling (FSS).

Chunk Self-Scheduling (CSS) It works like PDS, but instead of assigning iterations one by one, it assigns iterations in equally-sized chunks (Figure 2c) (FANG et al., 1990). Small chunk sizes deliver good load balancing, but they likely introduce prohibitive runtime overheads. In contrast, large chunk sizes avoid this problem, but may increase load imbalance. When the chunk size is fine-tuned, near-optimal load balancing is achieved (GRAHAM, 1969; FANG et al., 1990; BALASUBRAMANIAM et al., 2012), and when it equals to one, this scheduling strategy degenerates to PDS.

Guided Self-Scheduling (GSS) It also assigns chunks of loop iterations to threads on demand, but it dynamically changes their size at runtime according to the Load Imbalance Amortization Principle (POLYCHRONOPOULOS; KUCK, 1987). More precisely, the size of the next chunk is given by the number of remaining iterations divided by the number of threads (Figure 2d). The idea of having a decreasing chunk size is to offer a compromise between achieving good load balancing while reducing runtime overhead. Nevertheless, GSS may not deliver good performance when first iterations of a parallel loop are much more time-consuming than the iterations that follow.

Factoring Self-Scheduling (FSS) It works similarly to GSS, but it differs in the way that chunk sizes are determined (HUMMEL; SCHONBERG; FLYNN, 1992). To address the scenarios in which

GSS does not perform so well, FSS computes the next chunk size by dividing a subset of the remaining loop iterations (usually half) evenly among the threads (Figure 2e). FSS introduces no significant runtime overhead compared to GSS, and it may deliver better performance.

2.4 LOOP SCHEDULING IN OPENMP

OpenMP is an industry and academia standard API for parallel programming on shared-memory architectures (DAGUM; MENON, 1998). It is available for the C and Fortran languages, and allows one to exploit data and task parallelisms without introducing heavy changes to a target source code. To achieve so, OpenMP relies on (i) the Fork-Join parallel programming model, to easily manage thread creation and termination; and (ii) compiling directives, to reduce the efforts of an engineer of designing parallel code.

The Fork-Join model works as follows. When the application starts, a single thread, called the master thread, is launched, and executed sequentially until it reaches a parallel region. At this point, the master thread spawns a set of worker threads to concurrently execute the next instructions. Finally, in the end of the parallel region, all worker threads synchronize their activities and terminate. The master thread then progresses to the next sequential section and executes it until another parallel section is found, and the whole process starts over.

Parallel regions are created in OpenMP with the `omp parallel` directive. This directive instructs the compiler to link routines for starting and terminating working threads, and thereby effectively consists in an implementation of the Fork-Join model. Other directives are also available in OpenMP to enable programmers to fine-tune their solutions. For instance, the `task` directive allows threads to execute different instructions as independent tasks whereas the `for` directive instructs the compiler to parallelize the computation of a loop.

Snippet 2.1 presents a fragment of C code with OpenMP directives. In this example, we perform an adjoint convolution operation, which finds application in signal and image processing. This algorithm takes as input two arrays b and c, and it computes the pointwise multiplication of the two arrays. The `pragma omp parallel for` directive instructs the compiler that the iterations of the following `for` loop should be executed in parallel; and the `schedule` clause changes the

Snippet 2.1: Adjoint convolution in OpenMP.

```
1 double *adjconv(double *b, double *c)
  {
3   double *a = calloc(N*N, sizeof(double);

5   #pragma omp parallel for schedule(static)
    for (int i = 0; i < N*N; i++)
7   {
      for (int j = i; j < N*N; j++)
9       a[i] += F*b[j]*c[i - j];
    }
11
    return (a);
13 }
```

way in which loop iterations are scheduled (line 5). OpenMP offers built-in support for PSS, FSS and CSS. The OpenMP community pragmatically refers to them as Static, Guided and Dynamic, respectively. Therefore, we will refer to these strategies using the latter notation, unless otherwise stated.

3 RELATED WORK

In Section 2.3 we have presented the classical loop scheduling strategies and highlighted their main strengths. These solutions mostly rely on the on-demand scheduling and chunk-size tuning techniques to deliver a good performance to a broad class of applications. However, classical strategies show up to be suboptimal in irregular applications, thus motivating research efforts on this subject. Modern strategies aim at overcoming this barrier by exploiting data locality, features from the underlying platform and characteristics of the workload. In the paragraphs that follow, we briefly introduce the locality- and architectural-aware approaches. Then, we discuss in more detail different workload-aware loop scheduling strategies which are more related to our proposal.

Locality-aware loop scheduling strategies were first introduced by Markatos and Le Blanc (1994) to address memory-intensive applications. These strategies focus on making efficient use of the memory system hierarchy and the data locality principle, so that memory access contention is reduced and thus performance is improved (KEJARIWAL et al., 2009). In this approach, a common solution is to couple a deterministic loop iteration assignment policy with a work-stealing technique, thus enabling data locality to be exploited while load balancing is delivered (DURAND et al., 2013). Additionally, locality-aware scheduling strategies are often combined with thread-mapping heuristics to reduce even further contention in memory accesses (DING et al., 2013). Indeed, substantial performance gains can be observed when locality-aware strategies are correctly employed. However, when comes to achieving load balancing in compute-intensive irregular applications, alternative strategies are necessary, such as those based on platform intricacies or characteristics of the workload.

Architectural-aware scheduling strategies rely on features such as platform heterogeneity, availability and topology to deliver superior performance (WU et al., 2009). In this approach, a common heuristic is to apply some weighted static scheduling to initially distribute chunks of iterations to processors with different capacities, and then dynamically assign theses chunks to the threads running on each of these processors (YANG; CHENG; LI, 2005; WU et al., 2012). This idea yields to remarkable performance gains over classical loop scheduling strategies. Nevertheless, even better performance may be delivered when the workload of the application is also considered (BULL, 1998).

3.1 WORKLOAD-AWARE STRATEGIES

Workload-aware loop scheduling strategies consider some information about the workload to deliver cutting-edge performance and scalability to compute-intensive irregular applications. In this section we present some research efforts on this class of scheduling strategies.

Adaptive Weighted Factoring (AWF) Targeting time-step applications with irregular behaviors, Banicescu (2003) proposed AWF. In this strategy, the chunk size of a parallel loop is dynamically adapted after each step in the application. The newly computed chunk size is based on the performance of the threads during the previous step and on their accumulative performance during all the previous steps. To evaluate the performance of AWF, two in-house applications were studied: (i) Laplace's Equation Solver on an unstructured grid using Jacobi's Method; and (ii) N-Body Simulations using the Barnes-Hut algorithm. The PSS and FSS strategies were considered as baselines. Experiments were carried out on a loaded homogeneous cluster, and the results unveiled that AWF may achieve up to 46% and 33.9% better performance than the baseline strategies in the unstructured grid and N-Body applications, respectively. Due to the notable performance of AWF, extensions have been proposed to enable its use on non-iterative applications as well (CARIÑO; BANICESCU, 2008). Nevertheless, the enhanced version of this strategy presented a performance that is comparable to the one achieved by FSS.

History-Aware Self-Scheduling (HSS) To address a broader class of applications, Kejariwal, Nicolau and Polychronopoulos (2006) proposed HSS. Unlike AWF, HSS relies on statistical information collected offline via profiling to carry out a smarter scheduling. Based on this extra knowledge, at every scheduling round, HSS computes the amount of workload to be assigned to a thread, considering both the workload of previously executed iterations and their corresponding actual workloads. To assess the performance of HSS, kernels from the Standard Performance Evaluation Corporation (SPEC) Benchmarks were studied, and the FSS and AWF strategies were considered as baselines. Experiments were carried out on an in-house simulator, and the results unveiled that HSS may outperform baseline strategies from 5% to 18%.

Knowledge-Based Adaptive Self-Scheduling (KASS) Based on a similar offline profiling-guided approach to HSS, Wang et al. (2012) introduced KASS. This strategy works in two phases: a static partitioning phase, and a dynamic scheduling phase. In the first phase, a knowledge-based approach is used to partition iterations of the parallel loop into local work queues of threads, which makes the total workload to be equally distributed to the threads, approximately. In the second phase, based on the self-scheduling rule, every local work queue is partitioned into chunks with decreasing sizes. Each thread gets a chunk from its local queue to execute, and when it finishes the execution of all the chunks in its local queue, it steals chunks from other threads. To evaluate the performance of KASS, two scenarios were studied: (i) outermost loops with kernels extracted from the SPEC Benchmarks; and (ii) inner loops with the Over-Relaxation, Jacobi Iteration and Transitive Closure in-house kernels. The classical GSS, FSS, Trapezoid Self-Scheduling (TSS), Affinity Self-Scheduling (AFS) strategies were considered as baselines. Experiments were carried out on a Symmetric Multiprocessing (SMP) machine, and the results unveiled that for outermost parallel loops, KASS is from 4.8% up to 16.9% faster than the classical strategies. On the other hand, for the inner loop scenario, KASS achieves up to 21% better performance than AFS.

3.2 SUMMARY OF RELATED WORK

Locality-aware loop scheduling strategies may deliver good load balancing to memory-intensive applications. However, when comes to irregular applications with compute-intensive needs, cutting-edge performance may be achieved with architectural- and workload-aware strategies. The former class of scheduling strategies rely on features such as platform heterogeneity, availability and topology, and may yield to remarkable performance gains on heterogeneous platforms. Nevertheless, even better performance may be delivered when the workload of the application is considered as well.

Workload-aware loop scheduling strategies rely on this observation and indeed unveil superior performance. However, current scheduling strategies that are based on this approach still present some weaknesses that should be addressed. These drawbacks are summarized in Table 1 and further discussed next.

Table 1: Existing workload-aware strategies versus BinLPT.

Strategy	Workload Estimation	Sched. Technique	Availability
AWF	online, regression	amortization	none
HSS	offline, regression	amortization	none
KASS	offline, regression	amortization	none
BinLPT	user-supplied, flexible	adaptive chunk-size LPT rule	libGOMP

Workload Estimation Existing workload-aware strategies rely on profiling and statistical regression techniques. However, regression techniques are inherently designed to well-behaved workloads and profiling may be either time-consuming (offline) or impose a substantial overhead on execution time (online). Furthermore, current workload-aware loop schedulers are tightly coupled to their estimation techniques, and thus the HPC engineer is not free to employ the one that yields to the best workload estimation for the target parallel loop. For instance, AWF is restricted to time-step applications; and HSS and KASS are limited to irregular loops that feature well-behaved workloads.

Highly-Irregular Loops Current workload-aware loop schedulers rely on the Load Imbalance Amortization Principle (recall Chapter 2), *i.e.* they assume that the ratio between number of chunks of loop iterations and the number of working threads is high enough, so that there are enough chunks to even out the load imbalance in the irregular application. However, these strategies may achieve suboptimal performance on applications whose workload is not in accordance with this assumption (PENNA et al., 2016). Furthermore, even though existing workload-aware strategies do rely on workload estimations to partition the loop iteration space in several chunks, they actually lack in using this knowledge when scheduling chunks of iterations.

Integration and Availability Although several workload-aware strategies do exist, their source-code is not available for download nor they are shipped with some widely-adopted API, library or framework for parallel programming. Indeed, the integration of these strategies with irregular applications is not trivial (BANICESCU, 2003), thereby further restricting their use in practice.

Evaluation Related work on workload-aware loop scheduling strategies report no comprehensive evaluation in what concerns variations in the underlying workload of the target parallel loop. Unfortunately, this lack in performance analysis may yield to misleading conclusions about the actual performance of existing workload-aware loop schedulers.

To address the above weaknesses of existing related strategies, in this work, we present a new workload-aware loop scheduling strategy, which we named BinLPT. This strategy, relies on some user-supplied estimation about the underlying workload of the target irregular loop to achieve load balancing. Such estimation may be retrieved either from the problem structure or through online/offline profiling, thereby enabling maximum flexibility to the HPC engineer. Furthermore, BinLPT does not rely on the Load Imbalance Amortization Principle; it uses an adaptive chunking scheme; and it schedules chunks using a hybrid scheme based on the LPT rule and on-demand scheduling. Consequently, BinLPT is capable of achieving load balancing even in highly-irregular parallel loops. We integrated our workload-loop scheduling strategy into the OpenMP's runtime library of GNU Compiler Collection (GCC), a widely-used parallel programming API for shared-memory architectures. Our implementation is open-source and publicly available for download, thereby enabling any parallel application that relies on this programming abstraction to benefit from it.

In addition to our new workload-aware loop scheduling strategy, we deliver a comprehensive performance evaluation of BinLPT and the selected baseline strategies. We assessed different irregular workloads using several benchmarking techniques (simulation, synthetic kernel benchmarking and application kernel benchmarking), thereby delivering a detailed performance understanding of the studied scheduling strategies.

4 THE BINLPT LOOP SCHEDULER

In this chapter, we present our novel workload-aware loop scheduling strategy. First, we introduce the preliminary research on which we relied to devise BinLPT. Then we discuss the internals of BinLPT. Next, we detail our strategy algorithmically. Then we discuss about workload estimation. Finally, we briefly present the integration of our strategy into GCC's OpenMP runtime system, which is called libGOMP.

4.1 PRELIMINARY RESEARCH

To conceive BinLPT, we relied on two preliminary researches, both devised during the preparation of this master thesis: a design methodology for workload-aware loop scheduling strategies based on simulation (PENNA et al., 2016); and (ii) a proof-of-concept draft workload-aware loop scheduler named SRR (PENNA et al., 2017). In the next paragraphs, we briefly introduce each of them in turn.

The design methodology ships two tools, a Genetic Algorithm (GA) and a simulator. The former consists in a heuristic search which is guided by the Load Imbalance Equation (Chapter 2) and that enables the exploration of the solution space of the Loop Scheduling Problem. We relied on this tool to aid us on the design of SRR, the basis loop scheduler on which we built BinLPT. On the other hand, the latter tool is an event-driven, fast, and highly-accurate simulator for simulating the execution of parallel loops. This simulator enables the rapidly evaluation of scheduling strategies and we iteratively employed it in our research, first when designing SRR, and later when conceiving BinLPT. We present the internals of this simulator in Chapter 5, and also we report the performance results obtained with it in Chapter 6.

SRR relies on user supplied information about the underlying workload of the target parallel loop. Based on this information, the scheduling algorithm first sorts iterations according to their load, and then it assigns pairs of iterations to threads in a static way, following a round-robin scheme. Each pair is formed up with the iterations not yet assigned to threads that have the highest and lowest loads, so that in the end each thread is assigned to a workload that is near to the total average workload, if the workload follows a quasi-uniform distribution. For a detailed description of SRR, please refer to (PENNA et al., 2017).

4.2 OVERVIEW OF BINLPT

BinLPT operates in two phases to deliver load balance to irregular parallel loops, namely *chunk partitioning* and *chunk scheduling*. The heuristics used in each phase are depicted in Figure 3, and they are indeed the key features that enable the superior performance of BinLPT.

In the *chunk partitioning* phase, the primary goal is to split the iteration space into chunks so as to amortize load imbalance while minimizing the number of chunks that are produced. In this way, runtime scheduling overheads can be reduced and iteration affinity may be exploited efficiently. Indeed, this sub-problem could be optimally solved in pseudo-polynomial time using a dynamic programming algorithm for the Linear Partition Problem (SKIENA, 2008). Nevertheless, since loop ranges may grow asymptotically, the overhead incurred by this algorithm makes its use prohibitive. Therefore, to come up with a chunk partitioning having the previously stated features, BinLPT relies on a workload-aware adaptive technique that takes as input a user-supplied threshold k, for the maximum number of chunks to generate, and works as follows. First it computes the average load ω_{avg} for a chunk based on the workload information and k. Figure 3a outlines the overview functioning of BinLPT. In this example, two threads are considered to compute a parallel loop which overall load is 125; k is set to 4 and the average workload is $\omega_{\text{avg}} = 31.25$. Next, BinLPT uses a greedy bin packing heuristic that bundles into a single chunk the maximum number of iterations whose overall load does not exceed ω_{avg} (Figure 3b).

In the *chunk scheduling* phase, the goal is to come up with a chunk/thread assignment that minimizes load imbalance. Therefore, BinLPT relies on a hybrid scheduling scheme that works as follows. First, chunks are sorted in descending order according to their loads (Figure 3c). Then they are statically scheduled to threads using the LPT rule, which assigns the heaviest chunks to the least overloaded threads, and the whole process is repeated over and over again, until all chunks are assigned (Figure 3d). Next, threads are unblocked and start computing. Then, whenever a thread finishes computing all its chunks, a chunk that has not yet been processed is assigned to this thread, on-demand. Figure 3e shows the final thread/chunk assignment after applying on-demand scheduling.

The combination of static and on-demand scheduling techniques is able to handle the load imbalance created by both predictable and unpredictable phenomena. Static scheduling based on LPT ensures a

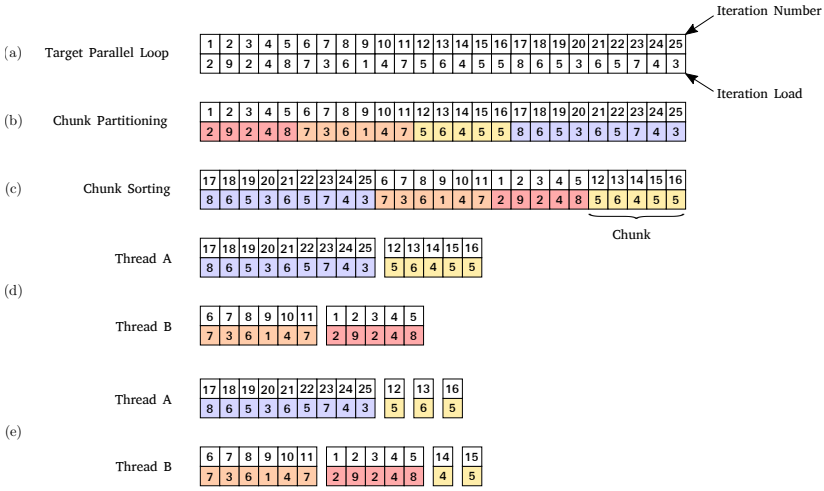


Figure 3: Example of loop scheduling using BinLPT: (a) target parallel loop; (b) chunk partitioning; (c) chunk sorting; (d) static scheduling based on LPT rule; and (e) on-demand scheduling.

4/3-approximation scheduling solution to the load imbalance incurred by the workload. On the other hand, on-demand scheduling ensures that unpredictable phenomena, such as communication latencies, external load interference, and poor workload estimation, are optimally tackled in a 2-approximative fashion (GRAHAM, 1969). At this point, it is worth noting that existing workload-aware loop scheduling strategies only rely on the latter technique (*i.e.* on-demand scheduling), even though they are aware of some information regarding the underlying workload for applying the former (*i.e.* LPT rule). Therefore, existing workload-aware loop scheduling strategies are inherently suboptimal, because they left the available information unexploited.

4.3 THE BINLPT ALGORITHM

The BinLPT loop scheduling strategy is outlined in Algorithm 1. It takes as input three parameters: an array that gives a load estimation of each iteration in the target parallel loop (A), the maximum number of chunks to generate (k) and the number of working threads (n). Then it returns a multiset (P) that states the thread/chunk assignment. In the pictured notation, T_j denotes the load of thread j; P_{T_j} the set of

chunks assigned to thread j; and $C \leftarrow C \cup \{\widehat{c_j}\}$ the inclusion of iteration A_i in the chunk $\widehat{c_j}$.

The algorithm starts by computing chunks according to the greedy bin packing heuristic detailed in the previous section (line 2). Then it sorts the produced chunks according to their loads (line 3). Next, chunks are statically scheduled following the LPT rule (lines 4 to 8). Later, during the execution, whenever a thread becomes idle, a new chunk that has not yet been processes is assigned on-demand to this thread, thereby accounting for unpredictable runtime phenomena.

With a smart implementation, this algorithm leads to low demands for both space and time. For the chunk computation, the workload estimation array (A) is traversed, thereby yielding a linear time cost ($O(n)$). For the chunk sorting, the time cost would grow logarithmically with the number of chunks if we were to use a sorting method such as Quick Sort ($O(n/k \log(n/k))$. Instead, sorting may be performed linearly with Counting Sort ($O(n/k)$) (CORMEN et al., 2001), if the workload estimates are within a known range. Finally, for the chunk scheduling, the time cost grows linearly with the number of chunks ($O(n/k)$). Therefore, the overall time cost for BinLPT is linear $O(n)$. The space complexity is bounded by the number of iterations to schedule and this is likewise linear ($O(n)$).

4.4 INTEGRATION WITH LIBGOMP

We implemented BinLPT in libGOMP, which is the OpenMP runtime system that comes with GCC. We made the enhanced version of this runtime system publicly available[1] under the GPL v3 License. This makes possible for any parallel application that is built on top of OpenMP to seamlessly use our scheduling strategy.

We introduced three changes to the mainstream libGOMP project. First, we hacked the `gomp_loop_runtime_start()` and `gomp_loop_init()` functions. The former invokes the runtime scheduler selected by the user, and we thus added the BinLPT strategy as a new possible option there. The latter function initializes the loop scheduler, and we inserted into it the BinLPT code corresponding to the chunk partitioning phase and the static chunk scheduling based on the LPT rule (Algorithm 1). Second, we added the `gomp_iter_binlpt_next()` function to the library. This function lookups the iteration/thread map output by the BinLPT strategy and effectively assigns iterations to the corresponding

[1] `www.github.com/lapesd/libgomp`

Algorithm 1 BinLPT loop scheduling strategy.

1: **function** BinLPT(A, k, n)
2: $C \leftarrow$ Compute-Chunks(A, k)
3: Sort(C, descending order)
4: **for** i **from** 0 to n **do**
5: $T_i \leftarrow 0$
6: $P_{T_i} \leftarrow \emptyset$
7: **for** i **from** 0 **to** $\mid C \mid$ **do**
8: $T_j \leftarrow \min\ T$
9: $P_{T_j} \leftarrow P_{T_j} \cup \{\widehat{C_i}\}$
10: $T_j \leftarrow T_j + \omega(\widehat{C_i})$
11: **return** P
12: **function** Compute-Chunks(A, k)
13: $j \leftarrow 0$
14: $C \leftarrow$ empty multiset
15: $\widehat{c_0} \leftarrow$ empty sequence
16: $\omega_{\text{avg}} \leftarrow \dfrac{\sum_{i_j \in A} w_j}{k}$
17: **for** i **from** 0 to $\mid A \mid$ **do**
18: **if** $\omega(\widehat{c_j}) > \omega_{\text{avg}}$ **then**
19: $C \leftarrow C \cup \{\widehat{c_j}\}$
20: $j \leftarrow j + 1$
21: $\widehat{c_j} \leftarrow (\widehat{c_j}, A_i)$
22: **return** C

threads. In addition, it also performs the on-demand chunk scheduling for those threads that have finished processing the chunks initially assigned to them. Finally, we provided a new runtime function named `omp_set_workload()`, which sets the workload information for the next loop. The BinLPT strategy relies on this information to run.

To invoke BinLPT, the programmer selects the `runtime` scheduler in the OpenMP `schedule` clause and set the OpenMP environment variable `OMP_SCHEDULE` to `binlpt,k`, where the last parameter controls the maximum number of chunks to be generated by BinLPT. Furthermore, the application should call the `omp_set_workload()` runtime function to inform the BinLPT scheduler about the load estimation of iterations in the next parallel loop.

Snippet 4.1 illustrates how BinLPT could be used in an adjoint convolution operation presented in Chapter 2. Note that in this kernel, we can estimate the load of each outter loop iteration based on how

many operations each of them will perform in the inner loop. Therefore, we fill up the `w` array accordingly to that information (lines 6 to 7), and next we pass this array to the BinLPT scheduler using `omp_loop_set_workload()` (line 9). Then, we invoke BinLPT by selecting the runtime scheduler (line 11).

Snippet 4.1: Adjoint convolution in OpenMP.

```
1 double *adjconv(double *b, double *c)
  {
3   double *a = calloc(N*N, sizeof(double));
    double *w = calloc(N*N, sizeof(double));
5
    for (int i = 0; i < N*N; i++)
7     w[i] = N*N - i;

9   omp_set_workload(w, N*N);

11  #pragma omp parallel for schedule(runtime)
    for (int i = 0; i < N*N; i++)
13  {
      for (int j = i; j < N*N; j++)
15      a[i] += F*b[j]*c[i - j];
    }
17
    return (a);
19 }
```

4.5 WORKLOAD ESTIMATION

Unlike other workload-aware loop scheduling strategies, BinLPT does not rely on a particular workload estimation technique. This allows the programmer to pick up one that best meets the requirements of the target parallel loop. Nevertheless, since the optimality of BinLPT relies on a static scheduling scheme, which is in turn highly-sensible to the workload estimation, we discuss next some alternatives for actually performing workload estimation along with their pros and cons.

The most straightforward approach is to instrument the target parallel loop and profile it, either offline or online. This solution is likely to lead to an accurate estimation whenever the workload does not larger varies from one execution to another (BULL, 1998). However, it is important to note that profiling may and time-consuming (BANICESCU; VELUSAMY, 2001). The second approach is to use compile-time analysis to estimate the load of each iteration. Indeed, this alternative may give a high-quality approximation for numeric parallel loops (THOMAN et al., 2012), but it fails when the irregularity arises from the input data itself. Finally, for some parallel loops, the load of each iteration may

be estimated from the problem structure itself, as a function of the input workload. Indeed, this approach may yield to the most accurate estimation, though its usability is application-dependent.

5 EVALUATION METHODOLOGY

In this chapter we present the evaluation methodology that we adopted in this work. First, we present the simulator and experimental kernels that we used to evaluate our strategy. Next, we introduce the performance metrics that we considered in our analysis. Finally, we detail our experimental design.

5.1 SIMSCHED: A LOOP SCHEDULER SIMULATOR

Simulation is an useful technique for (i) isolating the core variables of a given problem, (ii) precisely control and monitor the experimental environment, and (iii) evaluate and prototype solutions with minimum efforts. Unfortunately, at the time when this work was being carried out, no simulator for studying the Loop Scheduling Problem was publicly available. Therefore, we designed and implemented our own loop scheduler simulator (PENNA et al., 2016). We codenamed our tool SimSched, and we made it publicly available[1] under the GPL v3 License, so that other researchers could use it to design and evaluate new loop scheduling strategies.

Figure 4 presents an architectural overview of SimSched. Our tool carries out an event-driven simulation of a parallel loop execution based on three modules. *The Synthetic Workload Generator* module takes as input parameters (i) the number of loop iterations, (ii) the PDF associated to them, (ii) the load of iterations and (iv) the loop iteration shuffling seed; and it outputs a series of iterations with the given properties. Figure 5 illustrates how this happens. First, the frequency of loop iterations is generated according to a PDF (Figure 5a). Then each class of loop iterations is assigned to a different load (Figure 5b).

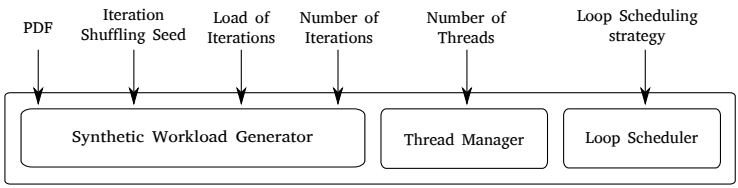
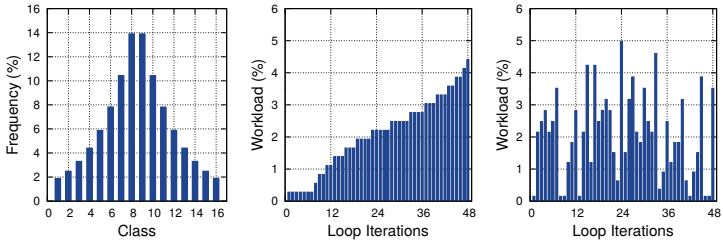


Figure 4: Architectural overview of the SimSched.

[1] Available at: `https://www.github.com/cart-pucminas/scheduler`



(a) PDF. (b) Load of iterations. (c) Loop shuffling.

Figure 5: Breakdown of synthetic workload generation.

Finally, the *Iteration Shuffling Seed* models how loop iterations are actually disposed in a for loop (Figure 5c). The *Thread Manager* module manages simulation threads. It takes as input parameter the number of threads to be created, and it schedules them in a round-robin fashion according to the selected scheduling strategy. This module uses two structures to manage threads: (i) a priority queue of running threads, which is ordered by their remaining workload to process; and (ii) a list of threads ready for execution. Finally, the *Loop Scheduler* module assigns iterations of the simulated parallel loop to simulation threads, according to the selected scheduling policy. The *Loop Scheduler Module* exports an interface that allows the simulation of both static and on-demand scheduling strategies.

SimSched can be easily extended to support new features. Nevertheless, its current version the following features are shipped:

- The *Synthetic Workload Generator* generates arbitrarly large workloads, following five different PDFs: Beta, Gamma, Exponential, Gaussian and Uniform. Loop iteration shuffling may be controlled by suplling a seed parameter.

- The *Thread Manager* outputs a detailed trace file, containing information about (i) the total load assigned to each thread; (b) the total number of chunks scheduled; and (c) performance metrics of the simulation (see Section 5.4).

- The *Loop Scheduler Module* implements the PSS, PDS, CSS, GSS, HSS and BinLPT strategies.

5.2 SCHEDBENCH: A SYNTHETIC KERNEL

While simulations in SimSched enable an upper-bound assessment of loop scheduling strategies, benchmarking with synthetic kernels would allow such analysis to be carried out in a real experimenting environment. Unfortunately, however, once again no such a tool was publicly available for use, thus forcing us to implement our own loop scheduler synthetic kernel benchmark. Likewise SimSched, we made this synthetic kernel publicly available[2] under the GPL v3 License, thus encouraging others to use it on future research.

Our synthetic kernel benchmark, entitled SchedBench, was implemented in OpenMP and it consists in an extension of the kernel proposed by Bull (1998). SchedBench (Algorithm 2) performs embarrassingly parallel computations on a private variable and thereby benchmarks the load balancing performance of a loop scheduler. This synthetic kernel takes six input parameters:

i. n: the number of loop iterations;

ii. w: the load associated to each iteration;

iii. s: the scheduling strategy to use;

iv. k: the number of threads;

v. l: the load of one operation in the synthetic kernel; and

vi. f: the computing complexity of the synthetic kernel.

First, the benchmark sets the loop scheduling strategy and number of working threads to use to s and n, respectively. Next it performs dummy computations in a parallel loop whose size equals n. Note that the number of operations that the synthetic kernel actually executes is proportional to w, l and f. The first parameter models the computing load associated to an iteration. The second parameter adjusts this load, so as it is costly enough to be benchmarked. On the other hand, the third parameter models the computing complexity of the kernel. That is, linear ($O(n)$), logarithmic ($O(\log n)$) or quadratic ($O(n^2)$). The more complex the kernel is, the stronger is the imbalance in the underlying workload of the parallel loop. This latter feature is indeed the extension that we added to the kernel proposed by Bull (1998), and thereby enables an even more comprehensive performance assessment of different loop schedulers.

[2] Available at: `https://www.github.com/lapesd/libgomp-benchmarks`

Algorithm 2 Synthetic benchmark.

```
 1: function SYNTHETIC-BENCHMARK(n, w, s, k, l, f)
 2:     set scheduler to s
 3:     set number of threads to k
 4:     parallel for i from 0 to n do
 5:         SYNTHETIC-KERNEL(w_i, l, f)
 6: function SYNTHETIC-KERNEL(w, l, f)
 7:     a ← 0
 8:     switch f do
 9:         case linear
10:             m ← w
11:         case logarithmic
12:             m ← w · log_2(w)
13:         case quadratic
14:             m ← w · w
15:     for i from 0 to m do
16:         for j from 0 to l do
17:             a ← a + 1
18:     return a
```

5.3 APPLICATION KERNELS

Although synthetic kernel benchmarking enables an upper bound performance analysis of loop scheduling strategies, this technique inherently lacks on delivering a realistic throughout assessment. For instance, data access patterns and instruction execution flows are too artificial. Furthermore, branch prediction, floating point and vectoring units, as well as instruction pipelining and thread synchronization mechanisms are not exercised. Therefore, as an attempt to fill this gap, we selected three application kernels to study. These kernels present great importance to the scientific community, span over different Dwarfs (ASANOVIĆ et al., 2006) and feature an irregular computation. In the next sections, we present each of these kernels.

5.3.1 MST

MST is a Graph Traversal application kernel that clusters a set of data points using the minimum spanning tree algorithm. Cluster strategies based on minimum spanning trees find applications in different fields, such as Cluster Analysis, Circuit Designing and Network-

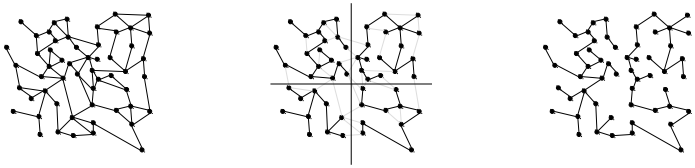
ing (GRYGORASH; ZHOU; JORGENSEN, 2006). The MST kernel works as follows. First, the Euclidean space is partitioned into several regions (Figure 6a). Then, Prim's Algorithm (PRIM, 1957) is executed on each of them (Figure 6b). In this computation, data is interpreted as a full graph, where data points are the nodes and point-to-point distances the edges. Thus, the resulting minimum spanning tree on each region locally clusters data according to the minimum Euclidean distance. Finally, minimum spanning trees are recursively merged, according to the minimum Euclidean distance, so data the final data cluster is produced (Figure 6c).

In this kernel, note that the cost for computing each local minimum spanning tree varies accordingly to the number of points that lie within each region. We relied on this observation to feed the workload-aware loop schedulers that we considered in our experiments.

5.3.2 LavaMD

LavaMD is an application kernel from Computational Fluid Dynamics (CFD) that performs N-Body Simulations. N-Body Simulations find applications in several scientific and engineering domains (SPRINGEL et al., 2005), and are frequently studied within the context of loop scheduling (BANICESCU; Flynn Hummel, 1995; BANICESCU, 2003; WANG et al., 2012). LavaMD was extracted from the Rodinia Benchmarks Suite (CHE et al., 2009), and it carries out a high-resolution simulation of the pressure-induced solidification of molten tantalum and quenched uranium atoms in a finitely-sized three-dimensional domain.

The LavaMD kernel works as follows. The 3D domain (Figure 7a) is decomposed into several equally-dimensioned n^3 boxes (Figure 7b. In this decomposition, any box has 26 adjacent boxes, except for boxes that lie within the boundaries of the domain, which in turn



(a) Initial data. (b) Partitioned data. (c) Clustered Data.

Figure 6: Computation performed by the MST kernel.

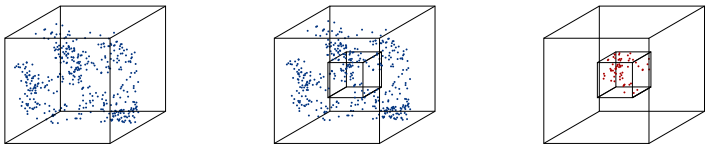
have fewer neighbours. At each computing step, particles interact only with other particles that lie within a user-specified cutoff radius, since ones at larger distances exert negligible forces (Figure 7c). Therefore, the box dimensions are chosen such that the cutoff radius of any particular box does not span beyond the boundaries of an adjacent box.

Note that the actual number of interactions to compute for a given particle α is proportional to the number of particles in the same box of α plus the total number of particles among all the boxes that surround α's box. We used this knowledge to estimate the computing load of each box when using workload-aware strategies in our experiments.

5.3.3 SMM

SMM is an application kernel from Sparse Linear Algebra that performs a multiplication between a sparse matrix and a dense vector. Besides finding applications in several scientific and engineering domains (BULUÇ et al., 2009), sparse matrix-vector multiplication is a frequently studied application kernel within the context of loop scheduling (WU et al., 2009, 2012). We extracted the SMM kernel from the Conjugate Gradient application from the NAS Parallel Benchmarks (NPB) (BAILEY et al., 1991).

In the SMM kernel, the sparse matrix is stored in compressed row format so that memory can be saved and data affinity exploited. The matrix is tiled in several blocks in a row-fashion, and next these blocks are processed in parallel. The actual number of floating point operations required to process each block varies accordingly to the number of non-zero elements in each block. Dense blocks, *i.e.* blocks with a lot of non-zero elements, are more costly to process than sparse blocks, blocks with a lot of zero elements. We relied on this observation to feed workload-aware loop scheduling strategies in our experiments.



(a) Physical domain. (b) Box decomposition. (c) Particle interactions.

Figure 7: Computation performed by the LavaMD kernel.

5.4 PERFORMANCE METRICS

To assess the performance of BinLPT we considered several performance metrics that are used in related work. Let n be the number of threads that execute a parallel loop, t_i the execution time of thread i, and W the total amount of work. Then, the following metrics may be computed (LUKE; BANICESCU; LI, 1998; CARIÑO; BANICESCU, 2008).

Parallel Time is the overall execution time of the parallel loop. Ideally, increasing the number of threads should exponentially decrease this metric.

$$\tau = \max\{t_i\}$$

Cost is the aggregate time spent to execute the parallel loop, and thus quantifies the waste of processor time. In an optimal scenario, increasing the number of working threads that execute the parallel loop should increase this metric by a constant factor.

$$\gamma = \tau \cdot n$$

Performance is the ratio of the total amount of work to the parallel time. Ideally, performance should scale up linearly with the number of threads.

$$\rho = \frac{W}{\tau}$$

Effectiveness is is the ratio of performance to cost, and thus it quantifies the throughput in contrast with resource waste. Ideally, when increasing the number of threads, effectiveness should scale up linearly.

$$\Gamma = \frac{\rho}{\gamma} = \frac{W}{n \cdot \tau^2}$$

Coefficient of Variance (CoV) is the ratio between the standard deviation and the mean execution time of the threads. Near-zero CoV suggests that there is no load imbalance, whereas a CoV near one indicates the opposite.

$$\lambda = \frac{\sigma\{t_i\}}{\mu\{t_i\}}$$

5.5 EXPERIMENTAL DESIGN

To evaluate the performance of BinLPT, we employed the simulator, as well as the synthetic and application kernels presented previously in this chapter. Overall we performed three sets of experiments, namely *Simulation*, *Synthetic Kernel Benchmarking* and *Application Kernel Benchmarking*; and Table 2 summarizes the parameters we considered in each of them.

The *Workload PDFs* are frequently studied in the context of the Loop Scheduling Problem (KRUSKAL; WEISS, 1985; BANICESCU; Flynn Hummel, 1995; BULL, 1998; BANICESCU; VELUSAMY, 2001; SRIVASTAVA et al., 2013). They were generated with the tool proposed in (PENNA et al., 2016) with a 16-point sampling precision, and with the following parameters: (i) Exponential $\gamma = 0.2$; (ii) Gaussian $\mu = 2.5$ and $\sigma = 1.0$; and (iii) Uniform $\alpha = 0.0$ and $\beta = 1.0$. The level of these parametes were chosen so that they would be consistent to the related work. The range of *Workload Shuffling Seed* in the *Simulation* was intentionally calculated to be big enough to yield to confidence intervals of 95%. The *Loop Scheduling Seed* for the *Synthetic Kernel Benchmarking* and *Application Kernel Benchmarking* were randomly chosen within this range. *Loop Sizes* and *Problem Sizes* were chosen so as to reflect the full processing capacity of the experimental platform (detailed later in this section). Baseline strategies were selected to be consistent with related works (BANICESCU, 2003; KEJARIWAL; NICOLAU; POLYCHRONOPOULOS, 2006). Guided and Dynamic loop schedulers are shipped with libGOMP by default, and thus we relied on this functionality. However, the HSS scheduler is not shipped with libGOMP neither is publicly available. Therefore, we implemented and integrated this strategy in libGOMP using the algorithmic description presented by Kejariwal, Nicolau and Polychronopoulos (2006). Chunk sizes were selected based on earlier experiments that revealed them to be the optimal values. In the next paragraphs, we describe each set of experiments in further details.

Table 2: Parameters for experiments.

Parameters	**Levels for Simulations**
Workload PDF	Exponential, Gaussian, and Uniform
Workload Shuffling Seed	{1...384}
Loop Size	{384, 768, 1152, .., 3072 }
Chunk Sizes	Guided {1}, Dynamic {1}, HSS {1}, BinLPT {384, 768, 1536}
Number of Threads	192
Parameters	**Levels for Synthetic Kernel Benchmarks**
Workload PDF	Exponential, Gaussian and Uniform
Workload Shuffling Seed	308
Loop Size	{384, 768, 1152, .., 3072 }
Kernel Load	2^{25} integer additions
Chunk Sizes	Guided {1, 2, 4}, Dynamic {1, 2, 4}, HSS {1, 2, 4}, BinLPT {384, 768, 1536}
Number of Threads	192
Kernel Complexities	Linear, Logarithmic and Quadratic
Parameters	**Levels for Application Kernel Benchmarks**
Workload PDF	Exponential, Gaussian and Uniform
Workload Shuffling Seed	308
LavaMD (Grid Size)	$11 \times 11 \times 11$
SMM (Matrix Size)	$(192 \cdot 2^{11}) \times (192 \cdot 2^{11})$
MST (Number of Points)	2^{22} $\mathbb{R}^2$
Chunk Sizes	Guided {2, 3, 4}, Dynamic {2, 3, 4}, HSS {2, 3, 4}, BinLPT {384, 576, 768 }
Number of Threads	{24, 48, 96, 120, 144, 168, 192}

Simulation: We used SimSched to carry out a state-space exploration of the Loop Scheduling Problem, and thus assess the performance of BinLPT in a great number of scenarios. We set the number of threads to 192 to reflect the full computational power of the experimental platform; and we adopted a full factorial experimental design, thereby resulting in 9216 scenarios for each strategy.

Synthetic Kernel Benchmarking: We used SchedBench to benchmark the overall performance of BinLPT in a real experimental environment, when varying the *Workload PDFs* and *Loop Sizes*. We additionally studied the impact of different *Kernel Complexities* in the performance of BinLPT. The levels for the latter pa-

rameter were chosen to match the complexity of widely studied algorithms. In this experiment, we also set the number of threads to 192 and adopted a full factorial experimental design, thereby resulting in 72 different scenarios for each strategy.

Application Kernel Benchmarking: We considered the three application kernels, for say LavaMD, MST and SMM, to evaluate the performance of BinLPT within real-world contexts. For this experiment, we adopted a fractioned experimental design, where we varied the number of threads from 24 to 192, with a constant step of 24. Overall, we studied 21 different scenarios for each scheduling strategy.

In the *Synthetic Kernel Benchmarking* and *Application kernel Benchmarking*, we carried out five replications of each possible configuration to account for the inherent variance of the measures in the experimental environment. For each replicate, the actual order in which individual runs were executed was randomly determined. This approach ensures that experimental results and errors are Independent and Identically Distributed (IID) random variables. In our experiments, the maximum relative standard deviation error (σ/μ) observed was below to 1.0%.

All experiments were carried out on a SGI Altix UV 2000 machine, which features 24 cache coherent NUMA nodes interconnected through SGI's proprietary NUMAlink6 (bidirectional). Each node has an Intel Xeon E5 Sandy-Bridge processor and 32 GB of DDR3 memory. Overall, this platform features 192 physical cores and 768 GB of memory. In our experiments, Simultaneous Multithreading (SMT) was disabled and we used a first-touch memory allocation strategy coupled with a compact thread affinity policy to mitigate runtime NUMA effects.

6 EXPERIMENTAL RESULTS

In this section we present and discuss the experimental results of our research. We will carry out this analysis in top-down fashion, first unveiling simulation results obtained with SimSched; then uncovering synthetic kernel benchmarking results in SchedBench; and finally detailing the experimental results observed for application kernels.

6.1 SIMULATION

We relied on simulations to assess the upper bound performance of BinLPT. In the sections that follow, we carry out an analysis on the *Workload PDF* and *Loop Size* parameters, and their impact on loop scheduling performance.

6.1.1 Workload PDF Breakdown

Figure 8 presents the load assigned to the slowest thread per PDF in the simulations when fixing the *Loop Size* in 768 iterations, and varying the *Workload Shuffling Seed*. Later in this section we present an analysis that considers variations in the *Loop Size*. In these plots, the median is evidenced, and the whiskers extend from each end of the box for a range equal to 1.5× the interquartile range. Furthermore, it is worth noting that y-axis starts at 0.5%. The rationale for this comes from the fact that in these experiments, the load assigned to the most overloaded thread should be at least $100/192 \approx 0.52\%$ (*i.e.* optimal solution for 192 threads).

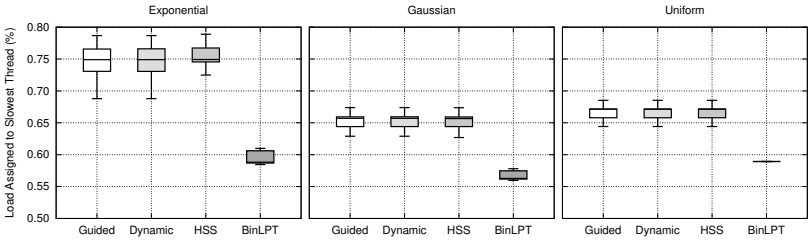


Figure 8: Assigned load per PDF in simulation.

Overall, the simulation results uncovered that BinLPT better balances the underlying workload of the target parallel loop, regardless the PDF type. When compared to the baseline strategies, our strategy may deliver from 1.14× (Gaussian PDF) to 1.27× (Exponential PDF) superior workload balancing. Furthermore, the results point out that there is no statistically significant difference between the baseline strategies themselves, in means of the of load that is assigned to the most overloaded thread. Finally, when observing the interquartile ranges, we noted that BinLPT presented the shortest one. This remark thus suggests that BinLPT is less sensitive to the way in which the underlying workload of the target parallel loop is shuffled.

6.1.2 Loop Size Scaling

Figure 9 depicts the load assigned to the most overloaded thread per *Workload PDF* in the simulations, when varying both the *Loop Size* and *Workload Shuffling Seed*. In these plots we only present results for the BinLPT and Dynamic strategies, once the Guided and HSS loop schedulers showed up statistically similar results to the Dynamic strategy. Furthermore, the lines in the plots picture the mean values of each strategy, with 95% of confidence. Finally, it is once again worth pointing out that y-axis starts at 0.5%, since the optimal solution for 192 threads surely is never below this value.

An overview analysis when scaling up the *Loop Size* unveiled that BinLPT yields to better load balancing than the other strategies. Significant performance gains were observed for *Loop Sizes* up to 1152 iterations, especially when considering Exponential workloads. Beyond 1152 iterations, the load balancing delivered by Dynamic showed up to be as good as our strategy, ranging from 10% worse to 3% better. The rationale for this behavior relies on the Load Imbalance Amorti-

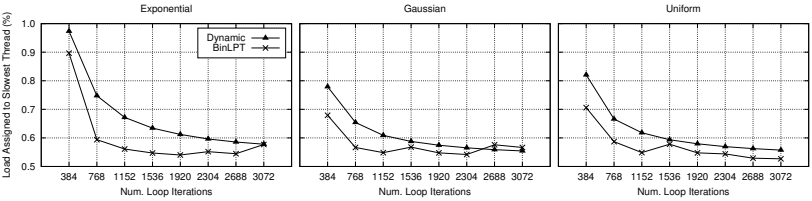


Figure 9: Simulation results for workload scaling.

zation Principle, which is indeed depicted in these plots. The more iterations there are in the parallel loop, the easier is to balance the underlying workload, and the smaller is the load imbalance in respect to the overall amount of workload. Therefore, unlike Dynamic and the other scheduling strategies, BinLPT may balance the underlying workload of a parallel loop even when the number of iterations is not sufficiently large to observe the Loop Amortization Principle. Based on this observation, simulation results thus suggest that BinLPT potentially outperforms existing strategies when the number of iterations of the target parallel loop is up to $8\times$ the number of working threads.

6.2 SYNTHETIC KERNEL BENCHMARKING

We employed the synthetic kernel benchmarking technique to uncover the potentials of BinLPT in real environments. In the next sections, we analyze the overhead scaling, as well as the impact of the *Loop Size* and *Kernel Complexity* on the performance of our loop scheduling strategy.

6.2.1 Overhead Scaling

The scheduling overhead of on-demand loop scheduling strategies depends on the number of chunks produced by a given strategy (CARIÑO; BANICESCU, 2008). The more chunks, the longer is the time wasted on synchronization structures and the poorer the iteration affinity is exploited. Therefore, the fewer chunks that are produced, the smaller the runtime overhead and thus the more scalable a given loop scheduler.

In this context, recall that the number of chunks that are produced by BinLPT and the considered baseline strategies may be somehow controlled. In the Guided and Dynamic strategies, the number of chunks that are generated depends on the size $|\widehat{x}|$ of the iteration space (*i.e. Loop Size*). In the former strategy, the number of chunks grows proportionally to $O(\log |\widehat{x}|)$, whereas in the latter strategy it grows with $O(|\widehat{x}|)$. In both strategies, the granularity of the chunk sizes may be further fine-tuned according to a parameter b. For instance, `Dynamic,1` will cause Dynamic to split the iteration space in unit-sized chunks ($b = 1$). On the other hand, `Guided,2` instructs Guided to generate chunks which are not smaller than 2 ($b = 2$). In the HSS and BinLPT

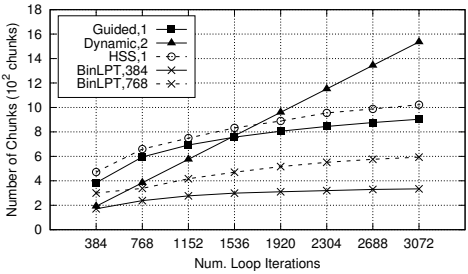


Figure 10: Number of chunks produced by scheduling strategies.

strategies, the chunk size dynamically adapts to the characteristics of the underlying workload, thus yielding to a floating number of chunks. Nevertheless, the granularity of the smallest chunk in HSS may be adjusted by a parameter; and the actual number of chunks produced by BinLPT may be fine-tuned by its k parameter. For instance, `HSS,2` will cause HSS to generate chunks that are not smaller than 2 iterations, likewise in Guided; and `BinLPT,288` instructs BinLPT to produce at most $k = 288$ chunks, pragmatically.

Figure 10 presents the number of chunks generated by each strategy for an Exponential-generated workload, when varying the size of the iteration space at a constant increase (384 iterations). We observed similar behaviors for the other workloads (*i.e.* Gamma- and Gaussian-based), and thus we omitted them. Overall, the results show that the number of chunks produced by BinLPT are far fewer than the ones produced by Guided, Dynamic and HSS loop schedulers. The number of chunks generated by Guided and HSS grows logarithmically, and for Dynamic it grows linearly; whereas for BinLPT it grows approximately with $(k \cdot \log |\widehat{x}|)/|\widehat{x}|$. In the performance analysis that follows, we unveil that small values used for k are enough for BinLPT to deliver superior performance than baseline strategies.

6.2.2 Kernel Complexity Breakdown

Figure 11 depicts the Performance (ρ) results per *Kernel Complexity*, when varying the loop scheduler strategy and their parameters and fixing the *Loop Size* in 1536 iterations, for an Exponential-generated underlying workload. In these plots, the median is evidenced, and the whiskers extend from each end of the box for a range equal to

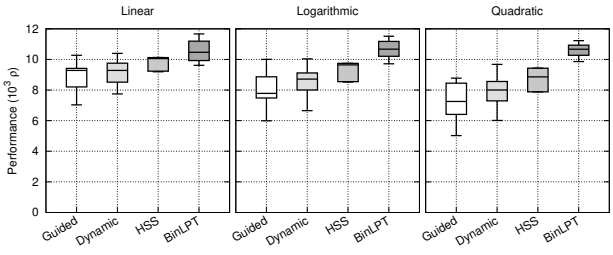


Figure 11: Breakdown of kernels for synthetic benchmarking.

1.5× the interquartile range.

An overview analysis uncovers two remarkable conclusions about BinLPT. First, our strategy leads to superior Performance (ρ) than the baseline strategies. Second, BinLPT delivers constant and more accurate Performance (ρ) when the kernel complexity increases. In contrast, baseline strategies are significantly impacted by the kernel complexity, thus suggesting that they suboptimally handle irregularity on the underlying workload of parallel loops. Indeed, BinLPT achieves optimal load balancing thanks to the LPT rule that is employed when scheduling chunks. The greatest Performance (ρ) gap was observed for a Quadratic kernel. In this scenario, BinLPT performed 1.77×, 1.27× and 1.22× better than Guided, Dynamic and HSS in mean values, respectively. We observed similar results for Gaussian- and Uniform-generated workloads and thus we omitted their respective plots.

6.2.3 Loop Size Scaling

Figure 12 presents Performance (ρ) results per *Workload PDF*, when fixing the *Kernel Complexity* in Quadratic and varying the iteration space (*i.e. Loop Size*). In these plots, synchronization overheads observed at run-time were filtered out; and each point pictures the best chunk size configuration for each strategy.

Overall, the results unveiled that BinLPT turns out to be the best loop scheduling strategy. The highest Performance (ρ) observed for BinLPT was for the scenario with 1536 iterations Exponential-generated workload, where it delivered at least 21.25% superior Performance (ρ) than the best configuration among all baseline strategies, for say `Dynamic,1`). On the other hand, the worst Performance (ρ) observed for BinLPT was for the scenario with 2304 loop iterations and

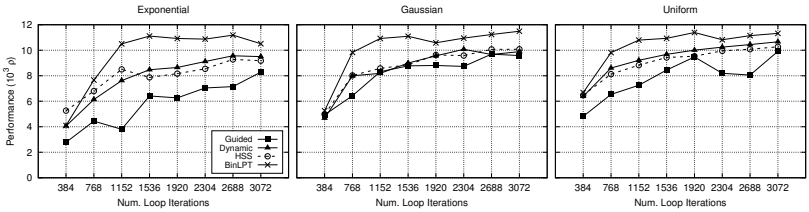


Figure 12: Synthetic benchmarking results for workload scaling.

Uniform-generated workload, where BinLPT presented 4.76% of performance gain than the best baseline strategies (`Dynamic,1`). Furthermore, these synthetic kernel benchmarking results further strengthen the findings that we uncovered with simulations in SimSched: (i) BinLPT delivers better performance to Exponential-generated workloads; and (ii) when the *Loop Size* increases, the Load Imbalance Amortization Principle is stronger, thus loop scheduling strategies should present similar performances asymptotically.

Nevertheless, at this point the following discussion is important. Although BinLPT did not perform significantly better, it turns out that, for this scenario, the weight of chunks generated by BinLPT was not fine-grained enough to amortize load imbalance. More precisely, BinLPT generated 1024 chunks, while Dynamic generated 2.25× more chunks. Putting it differently, a fair comparison between BinLPT and the other strategies should also account for the equivalence between the number of chunks generated by them and the number of chunks produced by our strategy. Recall that `BinLPT,k` splits the iteration space in at most k variable-size chunks, whereas `Dynamic,1` produces unit-sized chunks ($b = 1$). Therefore, for instance, a fair comparison would be in the scenario with 2304 iterations, between `BinLPT,2304` (at most 2304 chunks) against `Dynamic,1` (2304 chunks). When accounting for such observation, BinLPT may deliver up to 1.44× better Performance (ρ) than the best baseline strategy.

As a final remark, a note on the Performance (ρ) results observed for HSS is essential. Even though this workload-aware loop scheduler presented similar performance to the best baseline strategy (*i.e.* Dynamic), we recall that in order to drawn these plots we filtered out synchronization overheads observed at run-time. Overheads observed for Guided, Dynamic and BinLPT were far inferior than the execution time of the benchmark itself (*i.e.* less than 1%), thus not compromising conclusions from the previous performance analysis. However, for the

HSS scheduler we observed that synchronization overheads dominated execution time of experiments, thus yielding to effective Performance (ρ) results that were at least $2\times$ worse than the other strategies. The rationale for this behavior relies on the implementation of the loop scheduling strategies. Both Dynamic and Guided are shipped by default with libGOMP and rely on platform-dependent features and optimizations (*i.e.* atomic machine instructions) to mitigate synchronization overheads; and BinLPT is a lock-free loop scheduler. In contrast, HSS inherently relies on costly locking structures (*i.e.* spin-locks) thus causing significant contention. For this reason, for the remainder of this work, we will omit results for HSS, and carry out our analysis with Guided and Dynamic as baseline strategies.

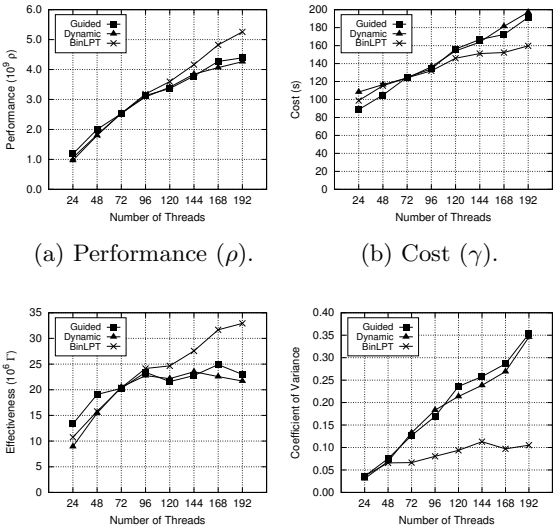
6.3 APPLICATION KERNEL BENCHMARKING

To analyze the performance of BinLPT in real-world scenarios, we employed the application kernel benchmarking technique. In the next sections, we discuss performance results that we observed in the three kernels that we studied: SMM, MST and LavaMD. We present a detailed analysis for the Exponential generated workload, because experimental results unveiled that BinLPT surpasses baseline strategies in this workload. Results for the Gaussian- and Uniform-generated workloads, are pictured in the appendices of this work.

6.3.1 SMM Kernel

Figure 13 presents experimental results for SMM kernel benchmarking. In these plots, we depict Performance (ρ), Cost (γ), Effectiveness (Γ) and CoV (λ) values for a Exponential-generated workload, when varying the number of working threads. These results picture the following scheduling configurations: `Guided,4`, `Dynamic,4` and `BinLPT,768`.

Recall that the SMM performs a sparse matrix by vector multiplication. This kernel presents a $O(mn)$ complexity and features significant affinity across the iteration space. Overall, BinLPT delivers better scalability to this kernel than baseline strategies, but performance differences were not highly expressive due to the low complexity of the kernel itself and the fine-grain scheduling employed in BinLPT. A more detailed analysis uncovered that the three loop scheduling strategies



(a) Performance (ρ). (b) Cost (γ).

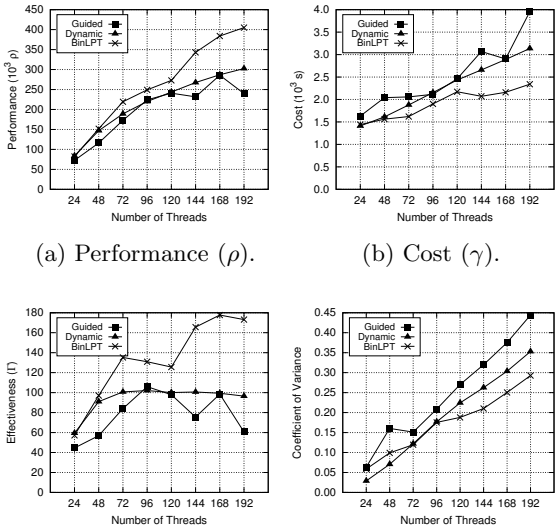
(c) Effectiveness (Γ). (d) CoV (λ).

Figure 13: Results for SMM kernel, Exponential workload.

present similar performance when up to 120 working threads are used. However, beyond this point, BinLPT stood out as the best strategy. The greatest performance difference was observed when the full processing capacity of the platform was used, with BinLPT delivering 15.78% better Performance (ρ) than baseline strategies (Figure 13a). Strong scaling capabilities of our strategy are further highlighted in the Cost (γ) and Effectiveness (Γ) results, with BinLPT delivering near-linear scalability in contrast to Guided and Dynamic (Figure 13b and Figure 13c). When analyzing CoV (λ) results, we observed that BinLPT clearly yields to slower variance values (Figure 13d). This observation further strengthens our hypothesis that BinLPT better balances the underlying workload of the target parallel loop among the working threads.

6.3.2 MST Kernel

Figure 14 presents experimental results for MST kernel benchmarking. In these plots, we present Performance (ρ), Cost (γ), Effectiveness (Γ) and CoV (λ) values for a Exponential-generated workload,



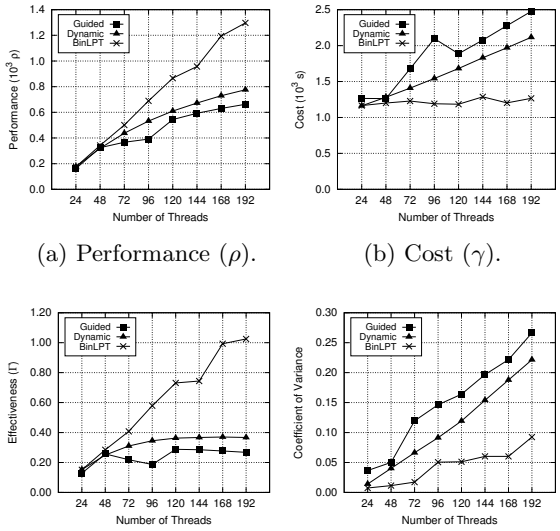
(a) Performance (ρ). (b) Cost (γ).

(c) Effectiveness (Γ). (d) CoV (λ).

Figure 14: Results for MST kernel, Exponential workload.

when varying the number of working threads. These results concern to the following scheduling configurations: `Guided,2`, `Dynamic,2` and `BinLPT,576`.

Recall that the MST clusters data using Prim's Minimum Spanning Tree algorithm. This kernel features a $O(v^2)$ complexity and an unpredictable memory access pattern, thereby introducing execution irregularities at run-time and favoring on-demand scheduling strategies (*i.e.* Guided and Dynamic). Overall, results unveiled that BinLPT delivers better performance and scalability than baseline strategies. Performance (ρ) values showed that when BinLPT faces a strong scaling scenario, it delivers quasi-linear scalability, and a maximum performance gain of 37.5% and 25.0% in contrast to Guided and Dynamic loop scheduling strategies, respectively. This behavior is also depicted in the Cost (γ) results (Figure 14b) and further evidenced in the Effectiveness (Γ) results (Figure 14c). Besides, it is worth noting that BinLPT presented a drop on Effectiveness (Γ) when using 96 and 120 threads due to a smooth increase on Cost (γ). The rationale for this behavior comes from the chunking heuristic of BinLPT itself. In these scenarios, the last chunks presented higher loads than the average, thus slightly increasing load imbalance and causing a negative impact on



(a) Performance (ρ). (b) Cost (γ).

(c) Effectiveness (Γ). (d) CoV (λ).

Figure 15: Results for LavaMD kernel benchmarking.

performance.

On the other hand, when analyzing CoV (λ) results, we have noticed that BinLPT presented a linear-growth behavior, likewise Guided and Dynamic. This outcome thus suggests that there is still room for improvement concerning load balancing of parallel loops featuring execution irregularities at run-time (*i.e.* irregular memory accesses).

6.3.3 LavaMD Kernel

Figure 15 presents experimental results for LavaMD kernel benchmarking. In these plots, we depict Performance (ρ), Cost (γ), Effectiveness (Γ) and CoV (λ) values for a Exponential-generated workload, when varying the number of working threads. These results concern to the following scheduling configurations: `Guided,2`, `Dynamic,2` and `BinLPT,768`.

Recall that LavaMD performs n-body simulations on a 3D domain. This kernel presents $O(n^3)$ complexity and features unbalanced CPU intensive computations. When analyzing the results, we observed that BinLPT delivers quasi-linear strong scaling Performance (ρ) scala-

bility and (Figure 15a) near constant Cost (γ) scalability (Figure 15d), in contrast to Guided and Dynamic that presented suboptimal behaviors, *i.e.* logarithmic Performance (ρ) and linear Cost (γ) scalability. This finding is further evidenced in the Effectiveness (Γ) plot (Figure 15c). When analyzinging this metric, we noted that BinLPT surpasses Guided and Dynamic and delivered up to 2.85× higher Effectiveness (Γ). Finally, when analyzing CoV (λ) results (Figure 15d), we found out that BinLPT led to lower execution time variance than Guided and Dynamic, thus showing that our strategy delivers better load balancing. Nevertheless, it is worth noting that BinLPT presented a stepped behavior, with steps occurring whenever the number of working threads perfectly divides the maximum processing capacity of the platform (*i.e.* 192 cores). This finding thus evidences the scenarios in which our strategy best uses the underlying platform (*i.e.* 72, and 168 working threads) and that BinLPT may be improved to overcome this strong scaling anomaly on CoV (λ).

6.4 SUMMARY OF RESULTS

In the previous sections, we presented a detailed analysis of the experimental results. In the paragraphs that follow, we present a summary of these results, highlighting our most outstanding conclusions.

Simulation: The workload analysis unveiled that BinLPT better balances the underlying workload of the target parallel loop, regardless the type of the workload. The highest gains over the baseline strategies were observed for Exponential-generated workloads, due to its inherently high irregular pattern. In this scenario, BinLPT delivered up to 1.27× superior load balancing, on average. Furthermore, BinLPT showed up to be the least sensitive strategy in what concerns the way in which the underlying workload of the target parallel loop is shuffled. On the other hand, the loop size scaling assessment uncovered that BinLPT achieves significant performance gains when the number of iterations of the target parallel loop is up to 8× the number of working threads.

Synthetic Kernel Benchmarking: The overhead scaling analysis pointed out that the number of chunks produced by BinLPT are far fewer than the ones produced by Guided, Dynamic and HSS loop schedulers. This observation suggests that BinLPT better exploits the iteration affinity while balancing the workload, in

contrast to baseline strategies. The kernel complexity benchmark uncovered that BinLPT performance gains over baseline strategies are strengthen by more complex kernels. The greatest Performance (ρ) gap was observed for a Quadratic kernel, when BinLPT performed up to 1.22× better than the best baseline strategy (HSS). Finally, the loop size scaling assessment validated our findings in the simulator: BinLPT surpasses baseline strategies when the Load Imbalance Amortization Principle is not strongly present. The highest Performance (ρ) observed for BinLPT was for the scenario with 1536 iterations Exponential-generated workload, where it delivered at least 21.25% superior Performance (ρ) than the best baseline strategy (Dynamic).

Application Kernel Benchmarking Strong scaling analysis uncovered that BinLPT surpasses in performance baseline strategies in all three kernels that were studied. Significant Performance (ρ) gains were observed for Exponential-generated workloads and when the full processing capacity of the experimental platform was used. In the SMM, MST and LavaMD kernels we observed up to 15.78%, 25.0% and 64.91% of performance improvement, respectively, in contrast to the best baseline strategy in each scenario.

7 CONCLUSIONS

To deliver high performance to large-scale engineering and scientific applications, particular intricacies of both the application itself and the underlying platform should be considered, so that tailored techniques can be employed. In this context, execution irregularity is an important feature that should be taken into account when scheduling tasks in a parallel application. While in regular applications, a static naive strategy that assigns an even number of tasks to the working threads may lead to an optimal solution, in highly irregular applications smarter heuristics are required to enable high performance. Indeed, task scheduling in multiprocessors plays an important role in HPC, and thus it is a recurring subject of research. For instance, this problem emerges when scheduling independent iterations of irregular parallel loops in shared-memory-based applications. In this context, the problem is referred to as the Loop Scheduling Problem, and it is reduced to the assignment of independent loop iterations such that (a) their load is evenly distributed, and thus execution time reduced; and (b) the scheduling overhead is minimized.

Several loop scheduling strategies have been proposed to address the aforementioned problem, and they mainly rely on on-demand scheduling and chunk-size tuning. When coupled together these techniques may deliver reasonable performance to a wide range of scenarios: the former dynamically handles load imbalance and runtime variations, whereas the latter mitigates scheduling overheads and enables iteration affinity exploitation. However, on-demand scheduling and chunk-size tuning do not consider any knowledge about the underlying workload of the target parallel loop, and thus scheduling strategies built upon them naturally turn out to be suboptimal. To address this limitation, workload-aware strategies were introduced. These strategies rely on some workload knowledge to adaptively fine-tune chunk sizes, and thus further amortize load imbalance and deliver superior performance. Nevertheless, existing workload-aware loop scheduling strategies lack on several points that still should be addressed, such as workload-estimation, chunk scheduling, and integration with applications.

To overcome the aforementioned weaknesses, in this work we proposed a novel workload-aware loop scheduling strategy called BinLPT. Our strategy is based on three features for delivering superior performance. First, it relies on some user-supplied estimation of the workload of the target irregular loop. Such estimation may be derived either from

the problem structure or through online/offline profiling, thus enabling maximum flexibility. Second, BinLPT uses a greedy bin packing heuristic to adaptively partition the iteration space in several chunks. The maximum number of chunks that may be produced is a parameter of our strategy, and it may be fine-tuned to better meet the characteristics of the irregular parallel loop. Third, it schedules chunks of iterations using a hybrid scheme based on the LPT rule and on-demand scheduling, thereby ensuring that load imbalance and runtime variations are optimally handled. Furthermore, to enable parallel applications to use our strategy, we integrated BinLPT into OpenMP and we made our implementation open-source and publicly available for download. In this way, parallel applications that rely on this programming abstraction may seamlessly benefit from BinLPT.

To evaluate the performance of BinLPT and deliver a through performance assessment to the scientific community, we embraced three techniques: (i) simulation; (ii) synthetic kernel benchmarking; and (iii) application kernel benchmarking. We employed simulations to assess the upper bound performances of our loop scheduling strategy under a great variety of workloads. Since there was no such a tool available for studying the Loop Scheduling Problem, we designed and implemented our own simulator. We codenamed it SchedBench and made it publicly available (PENNA et al., 2016). We used the synthetic kernel benchmarking to uncover the potentials of BinLPT in a realistic environment. We implemented an OpenMP version of the synthetic kernel proposed by (BULL, 1998) and assessed different algorithmic configurations of the target parallel loop. Finally, we considered application kernel benchmarking to analyze the effective performance of our strategy in practice. In this third assessment, we selected three different application kernels to study. These kernels present great importance to the scientific community, span over different domains and feature an irregular computation: Sparse Matrix by Vector Multiplication (SMM); Minimum Spanning Tree Clustering (MST); and N-Body Simulations (LavaMD). To deliver a rich comparative analysis, we considered as baseline in our experiments two classic loop scheduling strategies, namely GSS (OpenMP's Guided) and CSS (OpenMP's Dynamic), and one state-of-the-art workload-aware strategy, for say HSS.

Our experimental results consistently unveiled that BinLPT delivers superior performance than the baseline strategies, wether the assessment was made via simulation, synthetic kernel benchmarking or application kernel benchmarking. In the simulations, we observed that BinLPT better balances the underlying workload of the target

parallel loop, regardless the characteristics of the workload. BinLPT achieved the best performance on Exponential-generated workloads, delivering 1.27× superior load balancing than baseline strategies. In the synthetic kernel benchmarking, we observed that the potentials of BinLPT are further strengthen when the algorithmic complexity of the target parallel loop increases. Furthermore, BinLPT presented superior performance when scaling the size of the underlying workload, and it also stood out as the loop scheduling strategy with smaller scheduling overheads. Finally, application kernel benchmarking experiments uncovered that BinLPT features superior stronger scaling capabilities than baseline strategies for Exponential-generated workloads. In the SMM, MST and LavaMD kernels we noted up to 15.78%, 25.0% and 64.91% of performance improvement, respectively, in contrast to the best baseline strategy in each scenario.

8 FUTURE WORKS

In the paragraphs that follow, we highlight and briefly discuss related future works.

In what concerns SimSched, the simulator that we proposed to rapidly prototype and evaluate loop scheduling strategies, at least two works are possible. First, SimSched may be improved so that other variables such as processor availability, synchronization overhead and iteration affinity may be modeled. In this way, the accuracy of the simulator can be further increased. Second, our tool may be enhanced to support the simulation of NUMA machines heterogeneous platforms, such as those that feature hybrid CPU+GPU programming.

Regarding BinLPT, at several future investigations are possible. First, recall that the current implementation of our strategy relies on a chunking heuristic for bundling iterations. This heuristic favors performance over optimality, and it yields to low run-time overheads. On the other hand, a dynamic programming algorithm solves the same subproblem optimally, but it is more computing expensive. Therefore, a smarter chunking strategy would automatically select which solution to use, the current iteration bundling heuristic or the dynamic programming algorithm, based on the characteristics of the target parallel loop and the underlying workload. In a similar direction of adding autotuning support to BinLPT, a second future work would be to devise an heuristic for automatically selecting the maximum number of chunks to be produced by of our strategy (*i.e.* k parameter). Additionally, BinLPT should be improved to exploit iteration affinity, to deliver better performance to memory-intensive applications. Finally, BinLPT may be ported to other runtime environments such as TBB and Cilk; and APIs like OpenACC and OpenCL.

Lastly, with respect to performance evaluation of BinLPT, further investigations are valuable. Indeed, an ultimate performance analysis of our strategy would be to evaluate its integration within a real engineering application, such as one from Oil and Weather Forecast Industries. Nevertheless, an alternative is to monitor these applications, and execution trace files to feed our three-step evaluation methodology. In this way, experimental results would further uncover the performance potentials of BinLPT. Besides, yet another relevant work is to further expand the employed evaluation methodology to study other application kernels and explore other variations in the underlying workload of the target parallel loop.

REFERENCES

ASANOVIĆ, K. et al. **The Landscape of Parallel Computing Research: A View from Berkeley**. [S.l.], 2006. Available from Internet:: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>.

BAILEY, D. H. et al. The NAS Parallel Benchmarks: Summary and Preliminary Results. **International Conference on Supercomputing (ICS)**, p. 158–165, 1991. Available from Internet:: <http://doi.acm.org/10.1145/125826.125925>.

BALASUBRAMANIAM, M. et al. Towards the Scalability of Dynamic Loop Scheduling Techniques via Discrete Event Simulation. In: **International Parallel and Distributed Processing Symposium Workshops (IPDPSW)**. Shanghai, China: IEEE Computer Society Press, 2012. p. 1343–1351. ISBN 9780769546766.

BANICESCU, I. On the Scalability of Dynamic Scheduling Scientific Applications with Adaptive Weighted Factoring. **Journal of Cluster Computing (JCS)**, Kluwer Academic Publishers, v. 6, n. 3, p. 215–226, 2003. ISSN 13867857.

BANICESCU, I.; Flynn Hummel, S. Balancing processor loads and exploiting data locality in N-body simulations. In: **International Conference on Supercomputing (ICS)**. New York, USA: ACM Press, 1995. p. 43–es. ISBN 0897918169.

BANICESCU, I.; VELUSAMY, V. Performance of scheduling scientific applications with adaptive weighted factoring. In: **Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001**. [S.l.]: IEEE Computer Society Press, 2001. p. 791–801. ISBN 0-7695-0990-8.

BROQUEDIS, F. et al. ForestGOMP: An Efficient OpenMP Environment for NUMA Architectures. **International Journal of Parallel Programming**, Springer US, v. 38, n. 5-6, p. 418–439, 2010.

BULL, J. M. Feedback Guided Dynamic Loop Scheduling: Algorithms and Experiments. In: **International European Conference on Parallel and Distributed Computing (EuroPar)**. Southampton, United Kingdom: Springer, Berlin, Heidelberg, 1998. p. 377–382.

BULUÇ, A. et al. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In: **Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures - SPAA '09**. New York, New York, USA: ACM Press, 2009. p. 233. ISBN 9781605586069. Available from Internet:: <http://portal.acm.org/citation.cfm?doid=1583991.1584053>.

CARIÑO, R.; BANICESCU, I. Dynamic load balancing with adaptive factoring methods in scientific applications. **Journal of Supercomputing (Supercomputing)**, Springer US, v. 44, n. 1, p. 41–63, 2008. ISSN 09208542.

CHE, S. et al. Rodinia: A benchmark suite for heterogeneous computing. In: **International Symposium on Workload Characterization (IISWC)**. [S.l.]: IEEE, 2009. p. 44–54. ISBN 9781424451562.

CHEN, Q. et al. WATS: Workload-Aware Task Scheduling in Asymmetric Multi-Core Architectures. In: **International Parallel and Distributed Processing Symposium (IPDPS)**. [S.l.]: IEEE Computer Society Press, 2012. p. 249–260. ISBN 9780769546759.

CORMEN, T. H. et al. **Introduction to algorithms**. MIT Press, 2001. 1180 p. ISBN 0070131511. Available from Internet:: <http://dl.acm.org/citation.cfm?id=580470>.

DAGUM, L.; MENON, R. OpenMP: an industry standard API for shared-memory programming. **IEEE Computational Science and Engineering**, v. 5, n. 1, p. 46–55, 1998.

DINECHIN, B. de et al. A Clustered Manycore Processor Architecture for Embedded and Accelerated Applications. In: **International Conference on High Performance Extreme Computing (HPEC)**. Waltham, USA: IEEE Computer Society Press, 2013. p. 1–6. ISBN 978-1-4799-1365-7.

DING, W. et al. Locality-Aware Mapping and Scheduling for Multicores. In: **International Symposium on Code Generation and Optimization (CGO)**. Shenzhen, China: IEEE Computer Society Press, 2013. p. 1–12. ISBN 9781467355254.

DURAND, M. et al. An efficient OpenMP Loop Scheduler for Irregular Applications on Large-Scale NUMA Machines. In:

International Workshop on OpenMP (IWOPM). Canberra, Australia: Springer Berlin Heidelberg, 2013, (Lecture Notes in Computer Science, v. 8122). p. 141–155. ISBN 9783642406973.

FANG, Z. et al. Dynamic Processor Self-Scheduling for General Parallel Nested Loops. In: **IEEE Transactions on Computers (TC)**. [S.l.: s.n.], 1990. v. 39, n. 7, p. 919–929. ISBN 0271006080. ISSN 00189340.

FRANCESQUINI, E. et al. On the energy efficiency and performance of irregular application executions on multicore, NUMA and manycore platforms. **Journal of Parallel and Distributed Computing (JPDC)**, v. 76, p. 32–48, feb 2015. ISSN 07437315.

GAREY, M.; JOHNSON, D. **Computers and Intractability: a Guide to the theory of NP-Completeness**. New York, NY, USA: W. H. Freeman and Company, 1979. 338 p. ISBN 0716710447.

GAUTIER, T.; ROCH, J. L.; VILLARD, G. Regular versus Irregular Problems and Algorithms. In: **International Workshop on Parallel Algorithms for Irregularly Structured Problems (IRREGULAR)**. Lyon, France: Springer Berlin Heidelberg, 1995, (Lecture Notes in Computer Science, v. 980). p. 1–25. ISBN 978-3-540-44915-7.

GRAHAM, R. Bounds on Multiprocessing Timing Anomalies. **SIAM Journal on Applied Mathematics (SIPA)**, SIAM Press, v. 17, n. 2, p. 416–429, 1969. ISSN 0036-1399.

GRYGORASH, O.; ZHOU, Y.; JORGENSEN, Z. Minimum Spanning Tree Based Clustering Algorithms. In: **2006 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06)**. IEEE, 2006. p. 73–81. ISBN 0-7695-2728-0. ISSN 1082-3409. Available from Internet:: <http://ieeexplore.ieee.org/document/4031882/>.

HUMMEL, S.; SCHONBERG, E.; FLYNN, L. Factoring: a method for scheduling parallel loops. **Communications of the ACM**, ACM Press, v. 35, n. 8, p. 90–101, 1992. ISSN 0001-0782.

HURSON, A. et al. Parallelization of DOALL and DOACROSS Loops - A Survey. **Advances in Computers**, Elsevier, v. 45, p. 53–103, 1997.

KEJARIWAL, A.; NICOLAU, A.; POLYCHRONOPOULOS, C. History-Aware Self-Scheduling. In: **International Conference on Parallel Processing (ICPP)**. Columbus, USA: IEEE Computer Society Press, 2006. p. 185–192. ISBN 0769526365.

KEJARIWAL, A. et al. Efficient Scheduling of Nested Parallel Loops on Multi-Core Systems. In: **International Conference on Parallel Processing (ICPP)**. Ohio, USA: IEEE Computer Society Press, 2009. p. 74–83. ISBN 9780769538020. ISSN 01903918.

KRUSKAL, C.; WEISS, A. Allocating Independent Subtasks on Parallel Processors (TSE). **IEEE Transactions on Software Engineering (TSE)**, IEEE Computer Society Press, SE-11, n. 10, p. 1001–1016, 1985. ISSN 0098-5589.

LUKE, E. A.; BANICESCU, I.; LI, J. The optimal effectiveness metric for parallel application analysis. **Information Processing Letters**, v. 66, n. 5, p. 223–229, 1998. ISSN 00200190.

MARKATOS, E.; Le Blanc, T. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. **IEEE Transactions on Parallel and Distributed Systems (TPDS)**, IEEE Computer Society Press, v. 5, n. 4, p. 379–400, 1994.

PENNA, P. H. et al. Design Methodology for Workload-Aware Loop Scheduling Strategies Based on Genetic Algorithm and Simulation. **Concurrency and Computation: Practice and Experience (CCPE)**, 2016. ISSN 15320626.

PENNA, P. H. et al. Assessing the performance of the SRR loop scheduler. In: **International Conference on Computational Science (ICCS)**. Zurich, Switzerland: [s.n.], 2017.

POLYCHRONOPOULOS, C.; KUCK, D. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. **IEEE Transactions on Computers (TC)**, IEEE Computer Society Press, C-36, n. 12, p. 1425–1439, 1987. ISSN 0018-9340.

PRIM, R. C. Shortest Connection Networks And Some Generalizations. **Bell System Technical Journal**, v. 36, n. 6, p. 1389–1401, nov 1957. Available from Internet:: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6773228>.

SKIENA, S. S. **The Algorithm Design Manual**. 2nd. ed. [S.l.]: Springer, 2008. 730 p. ISSN 09574174. ISBN 9781848000698.

SPRINGEL, V. et al. Simulations of the formation, evolution and clustering of galaxies and quasars. **Nature**, v. 435, n. 7042, p. 629–636, jun 2005. ISSN 0028-0836.

SRIVASTAVA, S. et al. Predicting the Flexibility of Dynamic Loop Scheduling Using an Artificial Neural Network. In: **International Symposium on Parallel and Distributed Computing (ISPDC)**. Bucharest, Romania: IEEE Computer Society Press, 2013. p. 3–10. ISSN 2379-5352.

THOMAN, P. et al. Automatic OpenMP loop scheduling: A combined compiler and runtime approach. In: **International Workshop on OpenMP (IWOPM)**. Rome, Italy: Springer Berlin Heidelberg, 2012, (Lecture Notes in Computer Science, v. 7312). p. 88–101. ISBN 9783642309601.

TZEN, T.; NI, L. Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers. **IEEE Transactions on Parallel and Distributed Systems (TPDS)**, v. 4, n. 1, p. 87–98, 1993. ISSN 10459219.

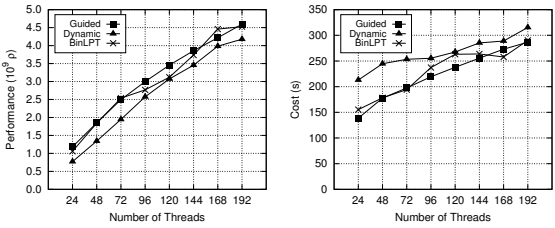
WANG, Y. et al. Knowledge-Based Adaptive Self-Scheduling. In: **International Conference on Network and Parallel Computing (NPC)**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, (Lecture Notes in Computer Science, 60973010). p. 22–32. ISBN 978-3-642-35606-3.

WU, C.-C. et al. Using Hybrid MPI and OpenMP Programming to Optimize Communications in Parallel Loop Self-Scheduling Schemes for Multicore PC Clusters. **Journal of Supercomputing (Supercomputing)**, Springer US, v. 60, n. 1, p. 31–61, 2009. ISSN 0920-8542.

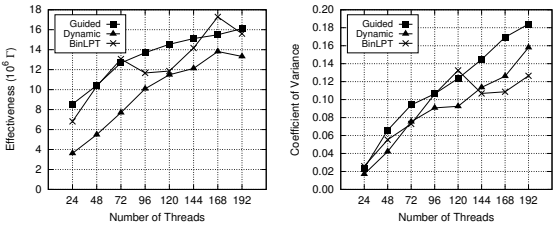
WU, C.-C. et al. Designing Parallel Loop Self-Scheduling Schemes Using the Hybrid MPI and OpenMP Programming Model for Multi-Core Grid Systems. **Journal of Supercomputing (Supercomputing)**, Springer US, v. 59, n. 1, p. 42–60, 2012.

YANG, C.-T.; CHENG, K.-W.; LI, K.-C. An Enhanced Parallel Loop Self-Scheduling Scheme for Cluster Environments. **Journal of Supercomputing (Supercomputing)**, v. 34, n. 3, p. 315–335, 2005.

APPENDIX A – SMM Kernel Results

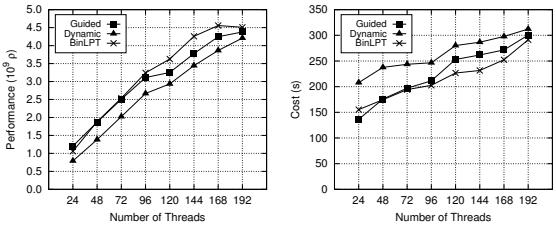


(a) Performance ($\boldsymbol{\rho}$). (b) Cost ($\boldsymbol{\gamma}$).

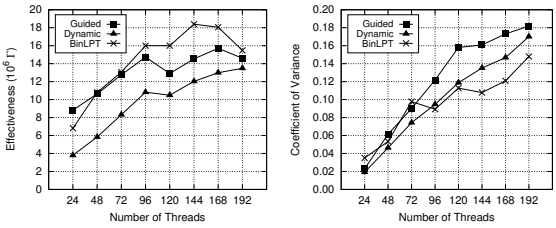


(c) Effectiveness ($\boldsymbol{\Gamma}$). (d) CoV ($\boldsymbol{\lambda}$).

Figure 16: Results for SMM kernel, Gaussian workload.



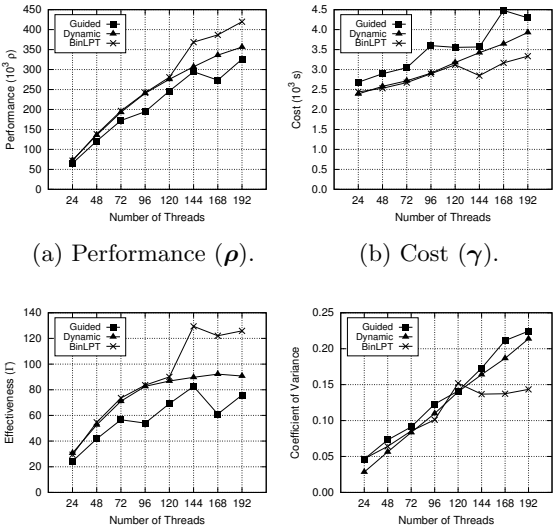
(a) Performance ($\boldsymbol{\rho}$). (b) Cost ($\boldsymbol{\gamma}$).



(c) Effectiveness ($\boldsymbol{\Gamma}$). (d) CoV ($\boldsymbol{\lambda}$).

Figure 17: Results for SMM kernel, Uniform workload.

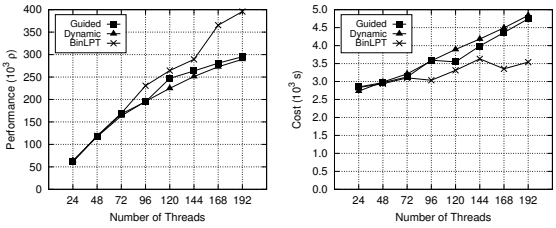
APPENDIX B – MST Kernel Results



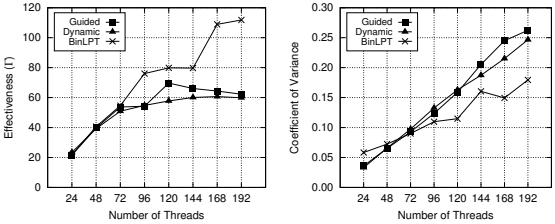
(a) Performance ($\boldsymbol{\rho}$). (b) Cost ($\boldsymbol{\gamma}$).

(c) Effectiveness ($\boldsymbol{\Gamma}$). (d) CoV ($\boldsymbol{\lambda}$).

Figure 18: Results for MST kernel, Gaussian workload.



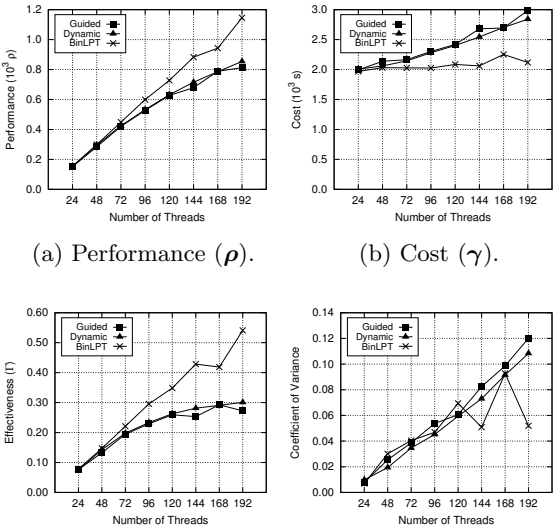
(a) Performance ($\boldsymbol{\rho}$). (b) Cost ($\boldsymbol{\gamma}$).



(c) Effectiveness ($\boldsymbol{\Gamma}$). (d) CoV ($\boldsymbol{\lambda}$).

Figure 19: Results for MST kernel, Uniform workload.

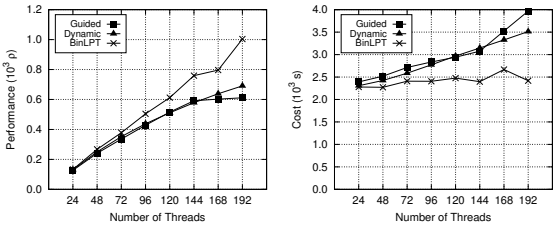
APPENDIX C – LavaMD Kernel Results



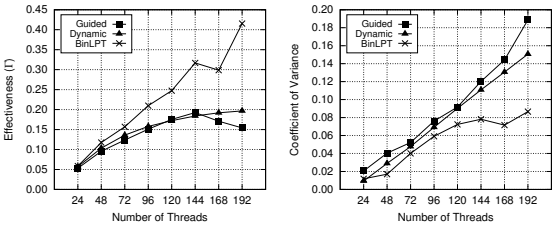
(a) Performance ($\boldsymbol{\rho}$). (b) Cost ($\boldsymbol{\gamma}$).

(c) Effectiveness ($\boldsymbol{\Gamma}$). (d) CoV ($\boldsymbol{\lambda}$).

Figure 20: Results for LavaMD kernel, Exponential workload.



(a) Performance ($\boldsymbol{\rho}$). (b) Cost ($\boldsymbol{\gamma}$).



(c) Effectiveness ($\boldsymbol{\Gamma}$). (d) CoV ($\boldsymbol{\lambda}$).

Figure 21: Results for LavaMD kernel, Uniform workload.

APÊNDICE D – Resumo Estendido em Português

D.1 INTRODUÇÃO

A comunidade de Computação de Alto Desempenho trabalha constantemente no projeto de soluções eficientes e escaláveis, visando suprir as demandas crescentes de aplicações científicas e industriais. Para tanto, essas soluções frequentemente endereçam características específicas da aplicações e da plataforma alvo, assim possibilitando que técnicas específicas possam ser aplicadas (FRANCESQUINI et al., 2015).

Nesse contexto, a irregularidade é uma característica relevante que afeta o desempenho de uma aplicação e, por isso, pode ser utilizada para classificar aplicações paralelas em dois grupos: *aplicações regulares* e *aplicações irregulares* (GAUTIER; ROCH; VILLARD, 1995). No primeiro grupo, a quantidade de computação necessária para solucionar um problema depende somente do tamanho dos dados de entrada. Por exemplo, no caso de um algoritmo de Multiplicação de Matrizes, o número de operações de ponto flutuante é proporcional ao tamanho das matrizes envolvidas na computação. Por outro lado, em aplicações irregulares, o tempo de execução também é significativamente impactado pelo conteúdo dos dados. Por exemplo, em simulações *N-Body*, o número interações entre partículas a serem computadas depende do número de partículas no sistema e também na distribuições das partículas no espaço.

Apesar de sutil, a diferença existente entre esses dois grupos impactam no projeto de soluções paralelas eficientes. Em aplicações regulares, a carga de trabalho pode ser trivialmente equi-particionada entre as *threads* de uma aplicação, simplesmente dividindo-se a carga total de trabalho pelo número de *threads*. Infelizmente, em aplicações irregulares, essa estratégia potencialmente conduz a um particionamento heterogêneo em termos de carga, assim causando desbalanceamento na execução da aplicações. Consequentemente, o tempo total da aplicação fica dominado pela *thread* mais sobrecarregada, implicando em problemas de escalabilidade.

D.1.1 Motivação

Distribuir igualmente a carga de trabalho entre as *threads* de uma aplicação, consiste em um problema de minimização NP-Difícil conhecido como o Problema de Escalonamento em Multiprocessadores (GRAHAM, 1969; GAREY; JOHNSON, 1979). Esse problema apresenta-se como um desafio para as comunidades acadêmica e industrial e

é um assunto recorrente na Computação de Alto Desempenho. Por exemplo, em aplicações paralelas de memória compartilhada, esse problema emerge no escalonamento de iterações de laços paralelos (FANG et al., 1990; POLYCHRONOPOULOS; KUCK, 1987; HUMMEL; SCHONBERG; FLYNN, 1992). Nesse cenário, o problema é referenciado como o Problema de Escalonamento de Laços e pode ser reduzido à atribuição de iterações independentes de um laço de forma que: (a) a carga de trabalho seja igualmente distribuída entre as *threads* que executarão o laço alvo; e (b) a sobrecarga de escalonamento é minimizada.

Dentre as diversas estratégias de escalonamento de laços foram propostas para endereçar o Problema de Escalonamento de Laços (KRUSKAL; WEISS, 1985; FANG et al., 1990; POLYCHRONOPOULOS; KUCK, 1987; HUMMEL; SCHONBERG; FLYNN, 1992; TZEN; NI, 1993; MARKATOS; Le Blanc, 1994; HURSON et al., 1997), podem-se destacar duas técnicas fundamentais. Na primeira, denominada escalonamento sob-demanda, iterações são escalonadas em tempo de execução, de forma que o desbalanceamento de carga e variações no ambiente de execução sejam gerenciadas. Na segunda técnica, denominada refinamento da granularidade de escalonamento, iterações são escalonadas em pacotes, denominados *chunks*, de forma que: (i) sobrecargas de escalonamento são minimizadas; (ii) o desbalanceamento de carga é amortizado; e (iii) afinidade entre as iterações é explorada. Quando utilizadas em conjunto, escalonamento sob-demanda e refinamento da granularidade de escalonamento podem conduzir a um desempenho razoável para uma larga gama de cenários. No entanto, essas técnicas não consideram a informação da carga de trabalho do laço paralelo alvo e por isso estratégias que se baseiam nelas revelam-se sub-ótimas (BALASUBRAMANIAM et al., 2012; PENNA et al., 2016).

Estratégias de escalonamento de laços cientes da carga de trabalho (ou *workload-aware*) foram introduzidas visando essa limitação (BANICESCU; Flynn Hummel, 1995; BULL, 1998; BANICESCU; VELUSAMY, 2001; KEJARIWAL; NICOLAU; POLYCHRONOPOULOS, 2006; WANG et al., 2012). Tais estratégias baseiam-se em alguma informação da carga de trabalho para adaptativamente ajustar a granularidade de escalonamento e assim amortizar ainda mais a irregularidade presente na carga de trabalho, conduzindo a um melhor desempenho.

D.1.2 Problemas

Apesar de estratégias de escalonamento de laços *workload-aware* proporcionarem melhores ganhos de desempenho que as demais estratégias, elas ainda enfrentam desafios que devem ser superados. Primeiro, essas estratégias baseiam-se em técnicas de regressão estatística e por isso são inerentemente projetadas para cargas de trabalho bem-comportadas. Além disso, é interessante observar que a estratégia de escalonamento e a técnica para estimar-se a carga de trabalho devem ser fracamente acopladas. Dessa forma, engenheiros podem adotar em suas soluções a técnica de estimativa mais adequada. Infelizmente, estratégias de escalonamento de laço cientes da carga de trabalho não oferecem essa flexibilidade.

Em segundo lugar, estratégias *workload-aware* falham em aplicar seu conhecimento sobre a carga do laço paralelo alvo ao escalonar *chunks* de iterações. Pragmaticamente, essas estratégias recaem sobre a técnica de escalonamento sob-demanda e assim, naturalmente, alcançam desempenho sub-ótimo. Além disso, a técnica de escalonamento sob-demanda conduz a problemas de escalabilidade, devendo portanto ser evitada.

Em terceiro lugar, estratégias *workload-aware* não foram avaliadas de maneira compreensiva até o momento, em especial no que diz respeito a variações na carga de trabalho. Essa lacuna de análise de desempenho deve-se a uma pobre disponibilidade de metodologias de avaliação e *benchmarks* que focam especificamente nesse cenário. Além disso, essa análise limitada compromete conclusões acerca dos reais potenciais de estratégias de escalonamento de laços *workload-aware*.

Finalmente, apesar da existência de diversas estratégias de escalonamento *workload-aware*, nenhuma delas encontra-se integrada em uma API, biblioteca ou *framework* de programação paralela publicamente disponível, como o OpenMP, Intel TBB ou Cilk. De fato, a integração dessas estratégias em aplicações irregulares não é trivial (BANICESCU; Flynn Hummel, 1995; BANICESCU, 2003), limitando ainda mais a sua aplicabilidade.

D.1.3 Objetivos e Contribuições

Tendo em vista os problemas enumerados anteriormente, o objetivo desse trabalho consiste em propor uma nova estratégia de escalonamento de laços *workload-aware*. Essa nova estratégia supera as

fraquezas mencionadas no que diz respeito à predição da carga de trabalho e escalonamento de *chunks*. Além disso, esse trabalho expõe uma implementação da estratégia proposta. Em resumo, o presente trabalho entrega as seguintes contribuições ao estado da arte:

- *Uma nova estratégia de escalonamento de laços intitulada BinLPT.* Para oferecer desempenho e flexibilidade superiores, essa estratégia baseia-se em três características. Primeiro, ela utiliza estimativas da carga de trabalho fornecidas pelo programador, que pode ser obtida a partir da estrutura do problema ou por *profiling online/offline*, permitindo máxima flexibilidade. Segundo, o BinLPT utiliza uma heurística gulosa para particionar adaptativamente as iterações do laço paralelo em diversos *chunks*. O número máximo de *chunks* produzidos é um parâmetro da estratégia proposta e pode ser ajustado para melhor adaptação das características do laço paralelo. Terceiro, o BinLPT escalona *chunks* à *threads* usando um esquema híbrido baseado na regra LPT e na técnica de escalonamento sob-demanda (GRAHAM, 1969), assim permitindo que variações no ambiente de *runtime* e irregularidades presentes na carga de trabalho sejam gerenciadas de forma ótima.

- *Uma integração do BinLPT no OpenMP.* O OpenMP é uma API de programação paralela para arquiteturas de memória compartilhada, que é utilizada amplamente pela academia e indústria (DAGUM; MENON, 1998). Essa implementação possui seu código aberto e está publicamente disponível para *download*, assim possibilitando que qualquer aplicação paralela que baseada nessa abstração transparentemente beneficie-se do BinLPT.

Além disso, visando entregar uma análise detalhada do BinLPT, uma avaliação extensiva baseada em simulação, *kernels* sintéticos e *kernels* de aplicações é apresentada. Os experimentos foram executados em uma máquina NUMA de larga escala (192 núcleos físicos) e o BinLPT foi estudado frente a diferentes cargas irregulares sintéticas. As estratégias `static` e `dynamic`, ambas disponíveis no OpenMP por padrão, foram utilizadas como base nesse trabalho.

O BinLPT foi projetado com base em duas pesquisas prévias, ambas desenvolvidas durante a preparação dessa dissertação: (i) uma metodologia de projeto para estratégias de escalonamento de laços ciente da carga de trabalho (PENNA et al., 2016); e (ii) uma estratégia preliminar denominada SRR (PENNA et al., 2017).

D.1.4 Estrutura do Trabalho

O restante desse resumo está organizado da seguinte forma. Na Seção D.2 a estratégia de escalonamento de laços proposta nesse trabalho é apresentada e detalhada. Na Seção D.3 uma síntese dos resultados experimentais e suas conclusões são expostas.

D.2 O ALGORITMO BINLPT

Nessa sessão, a estratégia de escalonamento de laços BinLPT será apresentada. Primeiro, as pesquisas prévias que possibilitaram o projeto dessa nova estratégia são introduzidos. Em seguida, os detalhes do BinLPT são discutidos. Por fim, a integração do BinLPT no ambiente OpenMP é apresentada.

D.2.1 Pesquisa Preliminar

O algoritmo BinLPT foi projetado com base em duas pesquisas prévias, ambas desenvolvidas durante a presente dissertação: (i) uma metodologia de projeto para escalonadores de laços (PENNA et al., 2016); e (ii) uma estratégia preliminar denominada SRR (PENNA et al., 2017). Nos parágrafos seguintes cada uma das duas pesquisas é sumarizada.

A metodologia de projeto propõe duas ferramentas, um algoritmo genético e um simulador. O primeiro consiste em uma heurística de busca, que é guiada pela Equação de Desbalanceamento de Carga (ver Capítulo 2), e possibilita a exploração do conjunto de estados de espaço do Problema de Escalonamento de Laços. Essa ferramenta foi empregada no projeto do algoritmo SRR, o escalonador de laços utilizado como base para criação do BinLPT. Por outro lado, a segunda ferramenta possibilita rápida avaliação de estratégias de escalonamento e foi empregada de maneira iterativa tanto no projeto do SRR quanto do BinLPT.

A estratégia SRR faz uso de estimativas da carga de trabalho fornecidas pelo usuário para realizar um escalonamento esperto das iterações de um laço paralelo alvo. Para tanto, primeiro as iterações do laço são ordenadas em forma crescente de acordo com sua carga e em seguida, são atribuídas em pares às *threads* em um esquema *round-robin*. Cada par é formado pelas iterações mais leve e mais pesada ainda não atribuídas, de forma que no final cada *thread* seja atribuída

uma carga de trabalho próximo à média. Uma descrição detalhada do SRR é apresentada em (PENNA et al., 2017).

D.2.2 Visão Geral do Algoritmo

O BinLPT opera em duas fases intituladas particionamento de *chunks* e escalonamento de *chunks*. As heurísticas empregadas em cada uma das fases são, de fato, as responsáveis por proporcionar ao BinLPT desempenho superior ao das estratégias relacionadas.

Na fase de particionamento de *chunks*, o objetivo é dividir o espaço de iterações em diversos *chunks* de forma que o desbalanceamento de carga seja minimizado ao máximo, com o menor número de *chunks*. Dessa forma, sobrecargas em tempo de execução podem ser reduzidas e a afinidade entre as iterações explorada de forma eficiente. De fato, esse particionamento pode ser resolvido em tempo pseudo-polinomial, utilizando-se um algoritmo de programação dinâmica para o Problema de Partição Linear (SKIENA, 2008). No entanto, uma vez que o número de iterações de um laço pode crescer assintoticamente, a sobrecarga desse algoritmo torna o seu uso proibitivo. Então, para contornar essa situação, o BinLPT emprega uma heurística adaptativa baseada na carga de trabalho. Essa heurística recebe como parâmetro um valor limite $\boldsymbol{k}$ e opera da seguinte forma. Primeiro, a carga média para um *chunk* ($\boldsymbol{\omega}_{\text{avg}}$) é computada, baseada nas estimativas da carga de trabalho e em $\boldsymbol{k}$. Em seguida, uma heurística gulosa empacota em *chunks* a máxima sequência de iterações cuja soma não excede $\boldsymbol{\omega}_{\text{avg}}$.

Na fase de escalonamento de *chunks*, o objetivo é produzir uma atribuição de *chunks* às *threads* de forma a reduzir o desbalanceamento remanescente. Para tanto, o BinLPT utiliza um esquema de escalonamento híbrido que funciona da seguinte forma. Primeiro, *chunks* são ordenados de forma decrescente de acordo com a sua carga. Em seguida, eles são estaticamente escalonados às *threads* utilizando a regra LPT, que atribui iterativamente o *chunk* mais pesado ainda não escalonado, à *thread* menos sobrecarregada. Na sequência, as *threads* são desbloqueadas e começam sua computação. A qualquer momento que uma *thread* termina de executar todos os *chunks* que haviam sido atribuídos a ela, um novo *chunk* é atribuído a essa *thread* sob-demanda.

Esse esquema de escalonamento híbrido possibilita que o desbalanceamento na execução tanto de fenômenos previsíveis quanto não previsíveis, sejam gerenciados de forma ótima. O escalonamento estático baseado na regra LPT garante uma solução **4/3**-aproximada para o des-

balanceamento proveniente da irregularidade da carga de trabalho. Por outro lado, o escalonamento sob-demanda entrega uma solução que é **2**-aproximada para o desbalanceamento oriundo de fontes aleatórias, como latências de comunicação, influência de carga de trabalho externa e estimativas não precisas sobre a carga de trabalho (GRAHAM, 1969). Nesse momento, é importante ressaltar que estratégias de escalonamento de laço *workload-aware* baseiam-se apenas na última técnica (*i.e.* escalonamento sob-demanda), apesar delas também considerarem uma informação da carga de trabalho e, assim, poderem aplicar a primeira heurística igualmente (*i.e.* regra LPT). Portanto, estratégias de escalonamento de laços existentes são inerentemente sub-ótimas.

D.2.3 Detalhes do Algoritmo

O algoritmo BinLPT (Algoritmo 3) recebe como entrada três parâmetros: um arranjo com as estimativas de cada iteração do laço paralelo alvo ($\boldsymbol{A}$), o máximo número de *chunks* a serem gerados pela estratégia ($\boldsymbol{k}$) e o número de *threads* ($\boldsymbol{n}$). Como saída, o algoritmo retorna um conjunto de conjuntos ($\boldsymbol{P}$) que enumera quais *chunks* foram atribuídas a quais *threads* na fase de escalonamento estático.

O algoritmo inicia particionando os *chunks*, segundo a heurística gulosa apresentada na seção anterior (linha **2**). Em seguida, ele ordena os *chunks* produzidos de acordo com sua carga de trabalho (linha **3**). Depois, os *chunks* são estaticamente escalonados às *threads* segundo a regra LPT (linhas **4** a **8**). Mais adiante, quando uma *thread* termina de executar todos os *chunks* que foram atribuídos a ela estaticamente, um novo *chunk* é atribuído a essa *thread* sob-demanda, permitindo assim que fenômenos não previsíveis possam ser gerenciados de forma ótima.

O algoritmo BinLPT possui requisitos baixos de espaço e tempo. Para o particionamento de *chunks* o arranjo com estimativas da carga de trabalho ($\boldsymbol{A}$) precisa ser percorrido, o que exige um custo linear ($\boldsymbol{O(n)}$). Para a ordenação de *chunks*, um tempo linear é necessário, uma vez que o algoritmo Count-Sort pode ser aplicado (CORMEN et al., 2001). Portanto, a complexidade de tempo do algoritmo BinLPT é $\boldsymbol{O(n)}$. Por outro lado, a complexidade de espaço está limitada ao número de iterações a serem escalonadas, sendo portanto igualmente linear.

Algorithm 3 Estratégia de escalonamento BinLPT.

```
1: function BinLPT(A, k, n)
2:     C ← Compute-Chunks(A, k)
3:     Sort(C, descending order)
4:     for i from 0 to n do
5:         T_i ← 0
6:     for i from 0 to | C | do
7:         T_j ← min T
8:         P_{T_j} ← P_{T_j} ∪ {ĉ_i}      T_j ← T_j + ω(ĉ_i)
9:     return P
10: function Compute-Chunks(A, k)
11:     j ← 0
12:     C ← empty multiset
13:     ĉ_0 ← empty sequence
14:     ω_avg ← (Σ_{i_j ∈ A} w_j) / k
15:     for i from 0 to | A | do
16:         if ω(ĉ_j) > ω_avg then
17:             C ← C ∪ {ĉ_j}      j ← j + 1
18:         ĉ_j ← (ĉ_j, A_i)
19:     return C
```

D.2.4 Integração com a libGOMP

O BinLPT foi implementado na libGOMP, o ambiente de *runtime* OpenMP do GCC. Essa implementação está disponível publicamente sob a licença GPL v3[1], assim permitindo que qualquer aplicação paralela escrita em OpenMP potencialmente beneficie-se do BinLPT.

Três alterações foram introduzidas na biblioteca libGOMP original. Primeiro, as funções `gomp_loop_runtime_start()` e `gomp_loop_init()` foram modificadas para reconhecerem o BinLPT como uma nova estratégia de escalonamento e invocar as rotinas do escalonador BinLPT propriamente ditas. Segundo, a função `gomp_iter_binlpt_next()` foi adicionada à biblioteca , que efetivamente efetua o escalonamento dos *chunks*. Por fim, uma função denominada `omp_set_workload()` foi incluída, para possibilitar que o programador informe a estimativa do laço paralelo que será executado.

Para invocar a estratégia BinLPT, o programador deve especi-

[1] `www.github.com/lapesd/libgomp`

ficar o escalonamento `runtime` na cláusula `schedule` no OpenMP e definir a variável de ambiente `OMP_SCHEDULE` para `binlpt,k`, onde o último parâmetro controla o máximo número de *chunks* a serem gerados pelo BinLPT. Além disso, a aplicação deve invocar a função `omp_set_workload()` para informar à estratégia BinLPT a estimativa de carga das iterações do laço paralelo que será executado.

D.3 RESULTADOS

Nos experimentos de simulação, uma análise da variação da carga de trabalho revelou que o BinLPT consegue alcançar um melhor balanceamento do que as estratégias contrastadas, independentemente do tipo de carga de trabalho. Os maiores ganhos observados foram para cargas de trabalho gerados a partir da função Exponencial, onde o BinLPT alcançou um desempenho em até **1.27**× superior, em média. Além disso, a estratégia proposta nesse trabalho mostrou ser menos sensível ao modo como a carga de trabalho está distribuída no laço. Por outro lado, uma análise da escalabilidade do tamanho do laço mostrou que ganhos significativos são alcançados com o BinLPT quando o tamanho do laço paralelo for até **8**× o número de *threads*.

Nos experimentos usando o *benchmark* sintético, uma análise da sobrecarga de escalonamento apontou que o número de *chunks* produzidos pelo BinLPT são substancialmente inferiores aos que os produzidos pelas estratégias consideradas como *baseline*. Essa observação sugere que o BinLPT é capaz de melhor explorar a afinidade entre as iterações, ao mesmo tempo em que efetua um balanceamento de carga mais efciente. Além disso, uma análise da complexidade de *kernel* revelou que os potenciais para o superior desempenho do BinLPT são reforçados à medida que a complexidade do *kernel* da aplicação aumenta. O maior desempenho alcançado em relação às estratégias consideradas como base foi **1.22**×. Por fim, um estudo de escalabilidade do tamanho do laço validou as demais conclusões obtidas a partir de resultados de simulação.

Nos experimentos usando *kernels* de aplicações, a análise de forte escalabilidade mostrou que o BinLPT supera as estratégias consideradas como base. Ganhos significativos foram observados para cargas de trabalho geradas a partir de uma distribuição Exponencial e quando a capacidade total da plataforma experimental era utilizada. Nos *kernels* SMM, MST e LavaMD, uma melhoria de até **15.78%**, **25.0%** e **64.91%** em relação às estratégias base foram observadas, respectivamente.